AIX Operating System for the PS/2 and System/370 Technical Reference

Volumes 1 and 2

Document Number SC23-2300-01

AIX Operating System
for the PS/2 and System/370

Technical Reference

Volumes 1 and 2

Document Number SC23-2300-01

Edition Notice

Third Edition (March 1991)

This edition applies to Version 1.2.1 of the IBM Advanced Interactive Executive for the System/370 (AIX/370), Program Number 5713-AFL, and for Version 1.2.1 of the IBM Advanced Interactive Executive for the Personal System/2 (AIX PS/2), Program Number 5713-AEQ, and to all subsequent releases until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for reader's comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, Department 52QA MS 911
Neighborhood Road
Kingston, NY 12401
U.S.A.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights or other legally protectible rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

Subtopics

Trademarks and Acknowledgements

Trademarks and Acknowledgements

The following trademarks apply to this book:

UNIX is a registered trademark of UNIX System Laboratories, Inc. i the USA and other countries. of AT&T in the United States of America and other countries.

AIX is a registered trademark of International Business Machine Corporation.

DEC VT100 and DEC VT220 are trademarks of Digital Equipmen Corporation.

Portions of the code and documentation were developed at th Electrical Engineering and Computer Sciences Department at the Berkeley campus of the University of California under the auspices of the Regents of the University of California.

IBM is a registered trademark of the International Business Machine Corporation.

Personal System/2 and PS/2 are registered trademarks of th International Business Machines Corporation.

System/370 is a trademark of the International Business Machine Corporation.

About This Book

This book describes the programming interface to the Advanced Interactive Executive (AIX) Operating System and provides reference information on AIX Operating System subroutines, system calls, file formats, and special files. Most of the facilities described apply equally well to either AIX PS/2 or AIX/370. When a facility is not common to both systems, it is so indicated.

Subtopics

Who Should Read This Book
What You Should Know
4.3BSD Compatibility
How to Use This Book
Related Publications

Who Should Read This Book

This book is written for experienced programmers who want to write application programs and systems software for the AIX Operating System.

What You Should Know

To use this book effectively, you should have an understanding of computer programming concepts and be an experienced C programmer. You should also be familiar with using AIX or UNIX System V commands, system calls, subroutines, file formats, and special files. If you are not already familiar with AIX or UNIX System V, see *Using the AIX Operating System* and the *AIX Operating System Commands Reference*.

If you have selected a language (through the **LANG** environment variable) which supports multibyte characters, you may need to know about information related specifically to the input of multibyte characters. Where differences from the base system (supporting US-English and European locales) exist, they are pointed out throughout this book.

4.3BSD Compatibility

AIX Release 1.2 incorporates many of the capabilities commonly available with 4.3BSD systems. However, the completeness and detail of the specific implementation may not be identical with that available in the latest BSD systems.

AIX integrates the BSD features which were available in 1987, although selected updates have been used to address particular issues. Specifically, AIX 1.2 does not include the 4.3BSD enhancements from the Tahoe and Mt. Xinu versions of BSD.

AIX is a cohesive blend of features derived from AIX/RT, System V, BSD, and NFS (from Sun Microsystems). All of these capabilities are compatible with the POSIX standard as defined in the *Portable Operating System Interface for Computer Environments (POSIX)*, IEEE 1003.1-1988. The details of implementation are in the following order of precedence:

1. POSIX compatibility
2. System V SVVS compatibility
3. AIX/RT compatibility
4. 4.3BSD compatibility.

To handle the conflicting demands of these different standards, AIX 1.2 makes design compromises, and in many cases, these conflicts are resolved with 4.3BSD.

This document accurately reflects the details of implementation chosen for the AIX product family.

How to Use This Book

This book has two volumes. Each volume contains a complete table of contents, list of figures, and an index for both volumes. Volume 1 gives you an overview of the various subsystems discussed in this book and describes the C language interface to the operating system calls. Volume 2 includes descriptions of the formats of various system and user files. It also introduces you to such facilities as text processing macro packages and describes the Advanced Display Graphics Support Library.

This book also contains four appendixes, including a glossary. Use the glossary to look up unfamiliar terms that are used in this book.

This *Technical Reference* includes all AIX Operating System facilities (system calls, subroutines, file formats, and special files). The facilities are presented in alphabetic order for easy retrieval. A detailed description of the use and function of each facility is given along with examples, where appropriate. In some cases, a given facility does not have a separate entry but is listed **under** a major facility. If you have difficulty finding a particular facility, look it up in the index.

Subtopics
Highlighting

Highlighting

This book uses different type styles to identify certain kinds of information. Following is a description of the various type styles that are used:

System call names, file names, and structure names are printed in **bold** type.

Variables and parameters are printed in **italic** type.

Constants appear in the descriptive text in UPPERCASE LETTERS

Information that you type or that appears on your display screen i printed in **monospace** type.

New terms introduced in the text are printed in ***boldface italic*** type. These terms are defined on first use in the text or in the glossary.

Related Publications

For additional information, you may want to refer to the following publications:

AIX C Language Reference, SC23-2058, describes the C programming language and contains reference information for writing programs in C language that run on the AIX Operating System.

AIX C Language User's Guide, SC23-2057, describes how to develop, link, and execute C language programs. This book also describes the operating dependencies of C language and shows how to use C language-related software utilities and other program development tools.

AIX Commands Reference, SC23-2292 (Vol. 1) and SC23-2184 (Vol. 2), lists and describes the AIX/370 and AIX PS/2 Operating System commands.

AIX Guide to Multibyte Character Set (MBCS) Support, GC23-2333, explains the basic concepts of AIX multibyte character set (MBCS) support and refers to other AIX books that contain more detailed information.

Managing the AIX Operating System, SC23-2293, describes such system-management tasks as adding and deleting user IDs, creating and mounting file systems, backing up the system, repairing file system damage, and setting up an electronic mail system and other networking facilities.

AIX Programming Tools and Interfaces, SC23-2304, describes the programming environment of the AIX Operating System and includes information about operating system tools that are used to develop, compile, and debug programs.

AIX Operating System TCP/IP User's Guide, SC23-2309, describes the features of TCP/IP and shows how to install and customize the program. It includes reference information on TCP/IP commands that are used to transfer files, manage the network, and log into remote systems.

AIX PS/2 INed, SC23-2001, shows how to use the INed editor to create, access, and store files. This book also includes reference information on INed commands and a listing of INed error messages.

AIX PS/2 INmail/INnet/INftp User's Guide, SC23-2076, describes the INmail/INnet/INftp/Connect programs and shows how to use these programs to send mail to and receive mail from local and remote computer systems. This book also shows how to transfer files to and from other computer systems installed on the network.

AIX PS/2 Keyboard Description and Character Reference, SC23-2037, describes the characters and keyboards supported by the AIX PS/2 Operating System. This book also provides information on keyboard position codes, keyboard states, control code points, code-sequence processing, and non-spacing character sequences.

Subtopics
Other Publications

Other Publications

Personal System/2 Hardware Interface Technical Reference, S68X-2330.

Table of Contents

Figures

1.0 Volume 1. System Calls and Subroutines

Subtopics
1.1 Chapter 1. AIX Operating System
1.2 Chapter 2. System Calls and Subroutines

1.1 Chapter 1. AIX Operating System

Subtopics

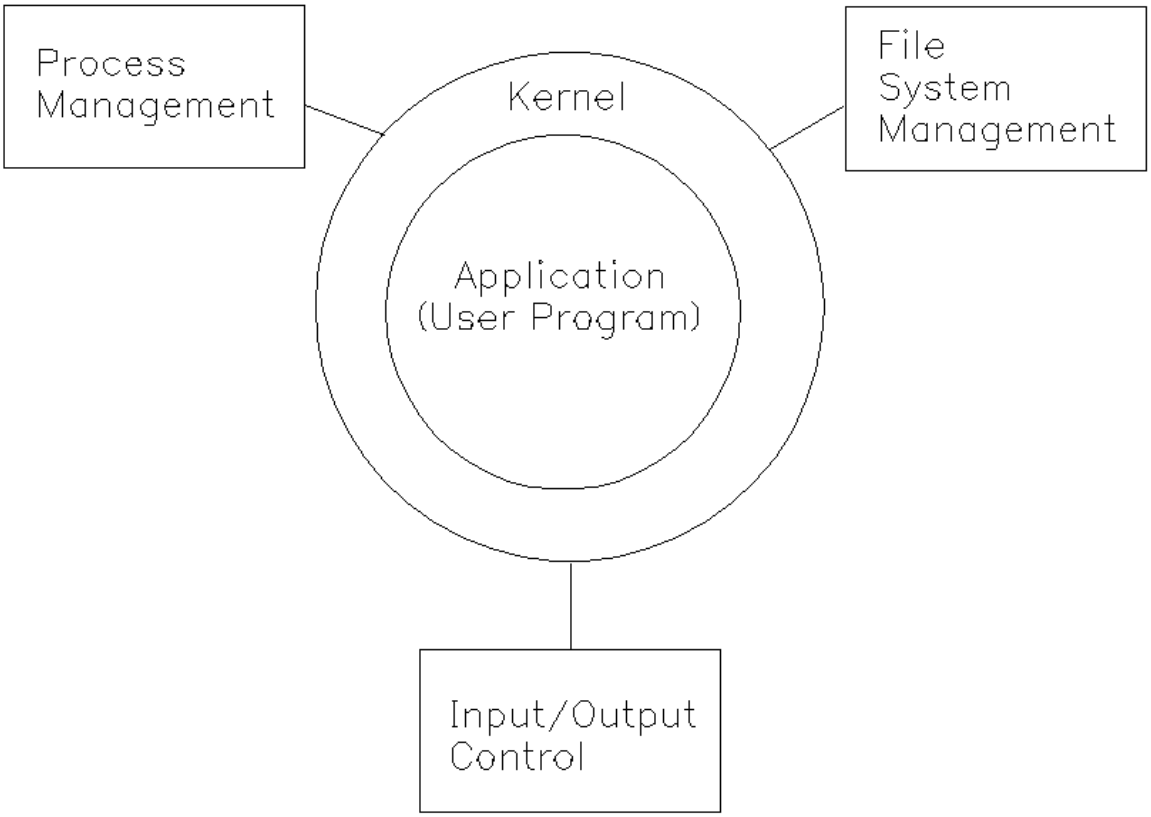
1.1.1 About This Chapter

This overview of the AIX Operating System is divided into three sections. The first section describes process management, creation, and scheduling. The second section is a description of the file system. The final section introduces I/O control and the I/O subsystem.

1.1.2 Kernel Functions and Structure

The ***kernel*** component of the AIX Operating System manages application program access to system-wide hardware and software resources. This component also provides an extended process environment and a usable and stable file system.

The kernel is mostly written in C language with some assembler language where necessary. The kernel provides services such as process control, file system management, input/output (I/O) control, and communication between processes and other miscellaneous facilities. Some of the kernel functions are discussed in sections that follow.



The kernel performs the following major functions:

File System Managemen

- File: open, close, read, write, change owner, get/change statistics, seek
- File system: mount, umount, get statistics
- Directory: change working directory, change root directory, make a directory, add or remove a link to a file
- Security: access permissions
- File replication: control where files are stored

Process Managemen

- Start and termination: fork a process, terminate this process, kill another process, kill a process group
- Set process group
- Informational: enable/disable accounting, get ID (process, parent,

group), get times
- Priority suggestion
- Wait for child process to terminate
- Lock data, text or stack in memory
- Execute a new program
- Signals: enable/disable signals, route signals to user routines, wait for a signal
- Semaphores: create semaphore, get semaphore ID, perform semaphore operations, delete semaphore

Input/Output Contro

- Device initialization: allocate, start, and terminate devices
- Device control: initiate and control transfer of data
- Interrupt handling: respond to device interrupts

Memory Managemen

- Private memory: grow, shrink
- Shared memory: create, attach, delete

Time Managemen

- Set time
- Get time

Resource Managemen

- Set and get user and group IDs
- Set and get user limits

Cluster Managemen

- Add or remove a site from the cluster
- Determine which sites are in the cluster
- Control where processes execute
- Determine the execution site of a process

1.1.3 Kernel Features

The kernel has the following features:

- Device Error Loggin
- File System Enhancement
- Virtual Memor
- Enhanced Signal
- Customization Facilities

The system commands and utilities are programs that operate in unprivileged mode and use system calls to the kernel resources to perform functions. System calls to the operating system kernel are utilized to assist in the completion of the function or to actually perform the function. The system commands and utilities are divided into several categories based on the type of service performed:

User Access Control	Controls user access to the system
System Status/Management	Provides system status
Program Development	Provides program development aids
Exchange Utilities	Provides exchange of files with other systems
Migration Aids	Provides data interpretation between systems
Information Handling	Provides data manipulation services
Communication	Provides intra-system communications services
Activity Monitoring and Accounting	Provides system trace and statistics
Directory Management	Provides directory manipulation services
File Management	Provides file manipulation services
Queue Management	Provides queue manipulation services
System Customization	Adds and deletes devices: Changes device information Displays configuration information

Subtopics
1.1.3.1 Bootstrap

1.1.3.1 Bootstrap

Before the kernel can run, it must be loaded into main memory of the machine. The bootstrap program is responsible for locating the kernel in the **<LOCAL>** file system, reading it into memory, and giving it control. The bootstrap program, **boot**, is loaded in by a smaller bootstrap program, **boot0**. **boot0** resides in the first 512 bytes of the active partition on the hard disk. This partition contains the VTOC (Volume Table of Contents) which describes the minidisks on the hard disk, a backup duplicate VTOC, the boot program itself, and a badblock minidisk (for ST506 systems only), in that order. This partition may also contain other minidisks.

With AIX PS/2, when the system is powered on or rebooted, the machine checks for a diskette in the first drive and attempts to boot from the diskette. If there is no diskette, the machine loads and executes the first sector (512 bytes) of the active partition of the first fixed disk. With AIX/370, the system always boots from the fixed disk. For AIX, this is the **boot0** program, which then finds and executes the bootstrap program. The AIX boot program, once loaded into memory, automatically loads **/unix.std** from the file system of the bootable AIX minidisk (the **<LOCAL>** file system resides on the bootable AIX minidisk). The bootstrap program gives the kernel control at its start entry point, thus completing the boot process. See "Creation and Execution" in topic 1.1.4.3.1 for additional information.

1.1.4 Process Control

A ***process*** is an instance of a program in execution. A ***program*** can be defined as an ordered set of instructions referred to as ***code*** and an execution environment including data and a stack.

A process as viewed by the operating system is the current state of the program that it is executing. This state includes a memory image (the logical layout of its parts in memory), the program text, program data variables used, register values, and other status of operating system resources (open files, pending signals, and so on) used by the process.

A process may be running a user program, an operating system utility, or part of the operating system. While a process is running or active, it may request services to be performed by the kernel. These services are typically initiated or performed in whole by the requesting process as it executes inside the kernel.

A process need not be resident in memory at all times. The kernel manages the physical memory, allocating and deallocating memory as required by multiple processes. A process image is said to be paged out of memory by having a portion of its memory image copied on a permanent storage device (disk). When there is insufficient physical memory, a process can be moved in whole or swapped to external storage.

Subtopics

1.1.4.1 User and Kernel Modes

An executing process can be either in ***user mode*** or ***kernel mode***. In user mode, a user program executes. In kernel mode, kernel code executes. When an executing user program (also called a user process) requires a function to be performed by the operating system, or needs to access system resources, it transfers into kernel mode by making a system call. A process in kernel mode has full control of the operating system. When the kernel (process executing in the kernel) completes the requested service, it usually returns to the user mode of the process.

A process in user mode can be preempted or taken out of execution at any time. In contrast, a process in kernel mode cannot be involuntarily preempted by another process. Thus, a process in kernel mode runs until it voluntarily relinquishes control of the processor (such as when waiting for a kernel resource or when starting an I/O operation) by relinquishing control to the kernel scheduler or by switching back to user mode (such as at the end of a system call).

The above generalities do not hold for ***interrupts*** and ***exceptions***. Interrupts are hardware or software signals that divert the processor to a special software routine. Exceptions or faults are unexpected events which are caused by a process.

All interrupts are serviced in kernel mode. If an interrupt occurs while the processor is executing a user mode process, a switch to kernel mode occurs for the duration of the interrupt service.

The most common form of interrupt is the device interrupt, also known as an I/O (input/output) interrupt. The system timer is a timing device which produces interrupts at fixed intervals. The interrupt routine for the system clock checks the priority and CPU usage of the processes and may preempt a process executing in user mode to implement a fair scheduling policy based on time slicing. This routine enables the processor to be shared fairly among many users.

Other device interrupts typically are serviced by device drivers that are kernel routines which perform device-specific processing. A typical device interrupt handler might post completion of I/O operations, start the next waiting operation, and notify the system that processes waiting for I/O completion can now be scheduled for further execution.

1.1.4.2 Memory Addressing

Memory management is performed by the AIX kernel. The AIX kernel and hardware support provide a paged virtual memory system. This means that each process can have separate virtual address space which is mapped into portions of physical memory pages. This allows each process to be much larger than the entire physical memory of the machine.

The virtual-to-physical address mapping is handled by the hardware memory management unit of the machine. When a process accesses virtual memory that is not associated with a physical memory page (not mapped), a page fault occurs which creates the necessary mapping; often by reading the processes' image from external storage. Page faulting, like other I/O operations, may cause the process to lose control of the CPU while it waits for I/O to complete. This allows the CPU to perform other non-I/O operations on behalf of other processes while I/O happens. (I/O is a magnitude slower than CPU operations). The memory required by the kernel is always mapped and never causes page faults. (On System/370s there may exist an additional level of address mapping when the machine is not configured V=R).

Subtopics
1.1.4.2.1 User Mode
1.1.4.2.2 Shared Segment
1.1.4.2.3 Kernel Mode

1.1.4.2.1 User Mode

A process in user mode accesses the following virtual segments while running. These segments are used to store information.

Text segment This segment is mapped and is addressable by a process in user mode. The ***text segment*** occupies the low addresses in the virtual address space of a process. This segment usually contains the user program code that executes. The information in this segment originates from the load module that executed an **exec** system call. (The **exec** system call is briefly discussed later). During execution, this segment is read-only, and a single copy of it is shared by all processes executing the same code.

Data segment This segment is mapped and is addressable by a process in user mode. The ***data segment*** of a user process begins on the logical boundary above the text segment. The process has read and write access to this segment. This segment is not shared by other processes and its size can be extended using a **brk** system call. The data segment contains an initialized portion used for data variables such as arrays, and a portion called **bss**, which is initialized to zeros.

Stack segment The stack segment is mapped and is addressable by a process in user mode. This segment of a user process starts at a high address in the process virtual address space and automatically grows in size toward the data segment as needed. This segment contains the run-time stack for a program, and user programs can write to it.

1.1.4.2.2 Shared Segment

In addition to the text, data, and stack segments that each process uses, a process can create and/or attach itself to segments that are accessible by other processes. These segments are called ***shared segments***. A set of system calls are available for using shared segments. When a shared segment is created or attached, the shared segment becomes part of the address space of the requesting process.

Shared segments can be used in either a read-only mode or in a read-write mode. There is no implicit serialization support when two or more processes access the same shared segment. If one process reads from a particular area of a shared segment, it is the responsibility of the two (or more) processes to coordinate their accesses to the shared area.

The load module may also specify additional text and data segments associated with shared libraries. Shared library text segments are just like ordinary text segments. Shared library data segments are initialized with data from the shared library, but the segments themselves are writable by the process and are not shared. The addresses at which the segments are mapped are specified in the shared library file.

1.1.4.2.3 Kernel Mode

The following areas are addressable by a process in kernel mode:

Process-specific memory (see below
All of the kernel code and data

The following areas contain all the data about a process needed by the kernel when the process is active:

Text Contains kernel program code that executes. It is read only by the kernel.

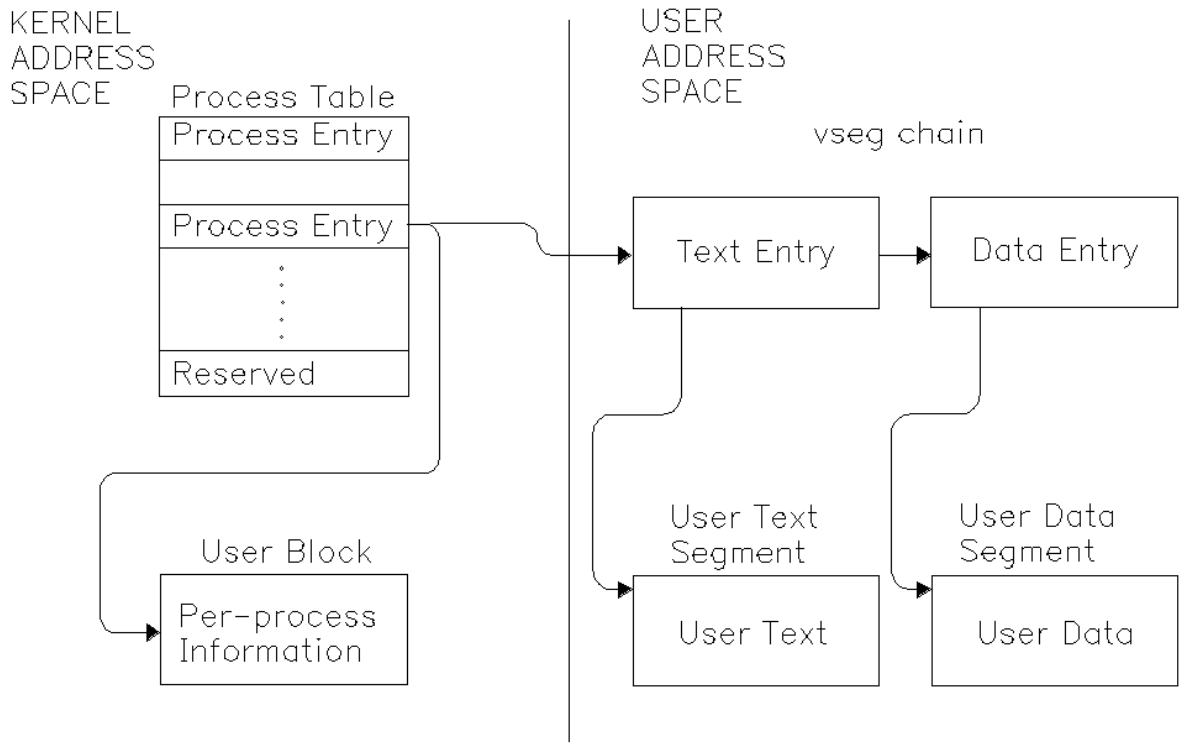
Global data Can be addressed by any process while in kernel mode. It contains tables, such as the open file table and process table, and other data, such as buffer pointers, maintained by the kernel.

Per-process data Sometimes called the ***user structure***, ***user area***, ***u.area***, ***user block***, or ***u.block***. It is a portion of the user process stack segment. This area is paged with the process. It contains process information, such as the current list of files opened by the process and arguments to the current system call. This information occupies the top of the kernel stack.

Kernel stack Paged with the user process. The kernel maintains a stack for each process. It saves the process information such as the call chain and local variables used by the kernel for the user process. The kernel stack is kept adjacent to the user block.

1.1.4.3 Process Data Structures

Most process management performed by the kernel is table searching and modification. The kernel maintains several tables to coordinate the running of many processes. The following diagram shows the tables maintained by the kernel to manage processes.



The ***process table*** contains an entry for each process that is created. This table contains the data needed when the process is not running. The structure of this table can be found in the **/usr/include/sys/proc.h** file in the file system. This table is always in memory so the kernel can manage events for the process. Each table entry details the state of a process. The state information includes the segment IDs of the process, the identification number of the process, and the identification of the user running the process. There is one table entry for each process; therefore, the number of processes that can be created is determined by the size of the table, which is specified as a customized parameter, **procs** in the **/etc/master** file. Process creation causes an entry in the process table and process termination frees an entry in the table. One table entry is reserved for a process with ***superuser*** authority. A process is recognized as superuser process and is granted special privileges if its effective user ID (UID) is 0.

Each process has its own copy of the unshared segments of the process, but some segments, such as text, can be shared. Sharing program text allows more effective use of memory. To keep track of shared and unshared segments, the system maintains a ***vseg table***. The structure of this table can be found in **/usr/include/sys/vseg.h**. A vseg table entry contains pointers to the page tables for the segment and the number of processes sharing this entry (which is 1 for unshared segments). When the number is reduced to 0, the entry is freed along with the segment. A vseg entry is allocated for each virtual segment of a process. The first process executing a shared text segment causes a new vseg entry to be allocated

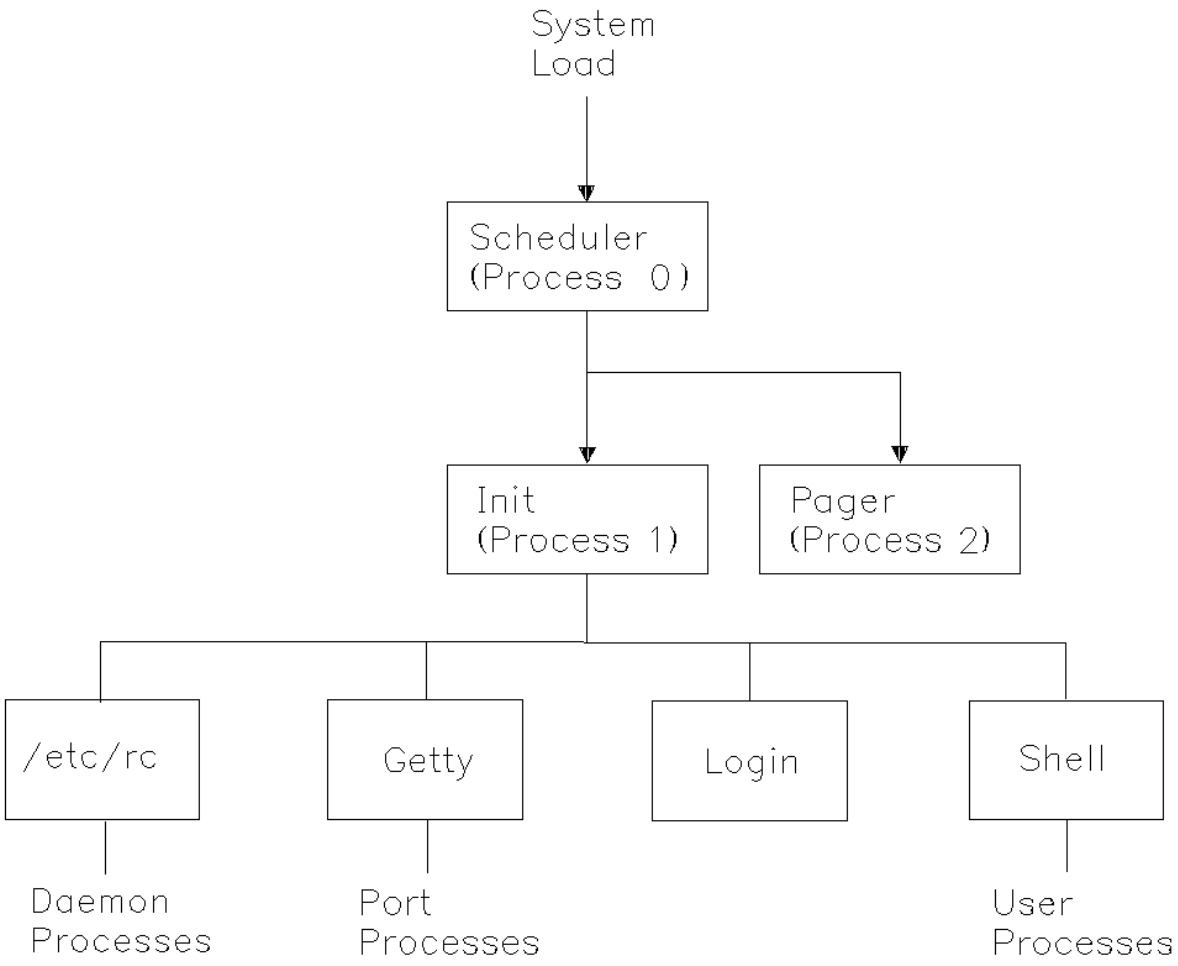
and the segment to be created. A second process executing an already allocated text segment causes the number in the vseg table to be incremented.

The ***user structure*** (also called per-process data area or user block) contains information that must be accessible while the process executes. One user structure is allocated for each active process. The user structure is directly accessible to the kernel routines. This structure can be found at **/usr/include/sys/user.h** in the file system. This block contains information such as user and group identification numbers for determining file access privileges, pointers into the system file table for the files opened by the process, and a list of actions to be taken for various signals. The user structure is part of the user stack segment. This chapter makes reference to entries in the user structure as **u.xxxx**, where **xxxx** is the structure member.

Subtopics
1.1.4.3.1 Creation and Execution
1.1.4.3.2 Parent and Child Processes
1.1.4.3.3 States of a Process

1.1.4.3.1 Creation and Execution

When the **/unix.std** file is found (see "Bootstrap" in topic 1.1.3.1), it is loaded and executed. First, it initializes disk data structures such as the free-list blocks, I/O buffer pool, the pool of character buffers, and the list of available inodes.



After the initialization is complete, the kernel starts to build the first process (process 0), also known as the swapper. The swapper is not created by the **fork** system call like other processes and it does not contain all the parts of a process. It is a unique process that contains only a data structure to be used by the kernel. Process 0 is the first entry in the process table and active only when the processor is in kernel mode. It is the first system process and is also responsible for scheduling. Process 0 creates the **init** process (process 1) and some special kernel processes by forking within the kernel. One of these kernel processes (process 2) is the **pager** (or page daemon) which is responsible for maintaining a supply of free memory pages. The remaining kernel processes support the Transparent Computing Facility.

Process 0, process 2, and the TCF kernel processes are incomplete process images, in that they contain no user code. Process 1 is the first completed process image and the ancestor of all subsequent processes. At this point no process will run until the swapper dispatches the first process ready to run. Process 1 executes an **exec** system call to overlay

itself with code from the **/etc/init** file.

As previously stated, all other processes are descendents of the **init** process. The **init** process first performs file system checks on the root and **<LOCAL>** file systems. Its subsequent operation is controlled by the file **/etc/inittab**, which runs the system startup scripts and controls the multi-user mode. The startup scripts are responsible for performing integrity checks, doing any necessary cleanup, mounting the normal file systems, enabling standard ports. After system startup runs successfully, the **init** process creates a **getty** using the **fork** system call for each enabled port specified in the **/etc/inittab** file. The **init** process performs the **exec** system call to **getty** which determines appropriate terminal speeds and modes by consulting the file **/etc/ports**. The **getty** program performs the **exec** system call to **login** to validate the user's password, and set the user ID (UID), the group ID (GID), the current directory, and so on. The **login** program executes the shell or the program specified in the **/etc/passwd** file as the first program to be run after **login**. The shell runs in the same process created by **init**. The shell performs the **fork** system call, which creates new processes for every command. While the system is running, the **init** process sleeps waiting for the termination of any of its child processes. When a user logs off, **init** creates a new **getty** via a **fork**.

1.1.4.3.2 Parent and Child Processes

A process can, for various reasons, create a copy of itself by issuing the **fork** system call. When this occurs, the original process is called the ***parent*** process and the newly created process is called the ***child*** process. The major difference between the original process, the parent, and the created process, the child, is that they have different process identification numbers, parent process identification numbers, and time accounting information.

The **fork** system call causes the total number of system processes to increase by causing a new process, the child, to be created. Besides the differences mentioned previously, each receives a different value from the **fork** system call. (The child receives the value 0 and the parent receives the ID of the child process.) The two processes share open files and each process can determine whether it is the parent or the child by the value received. The parent may or may not wait for any of its child processes to terminate.

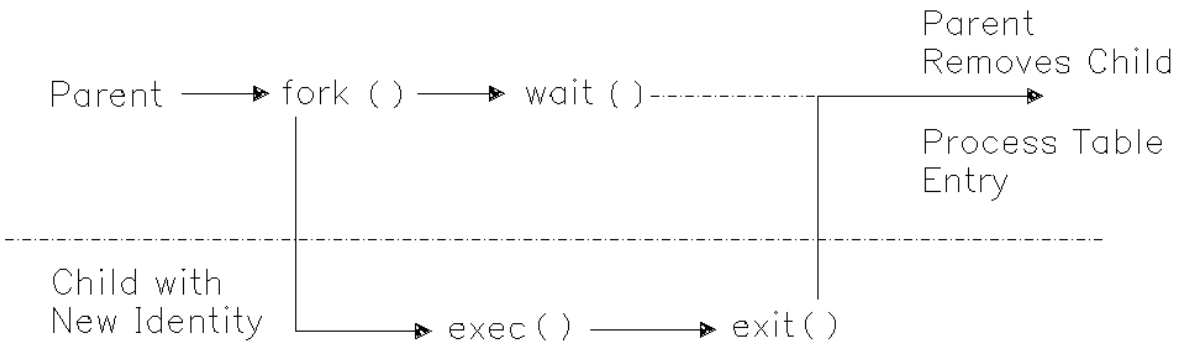
The **exec** system call causes the process to overlay the information it contains with new information. During an **exec** system call the process exchanges current text and data segments for new data and text segments. The total number of system processes does not change, only the process that issued the **exec** is affected. After the **exec** system call, the process identification number is the same and open files remain open (except close-on-exec files).

The **exit** system call terminates the process that issued the **exit**. All files accessed by that process are closed and the waiting parent is notified. A ***zombie*** process is a terminated process whose entry remains in the process table. The parent process is responsible for clearing the entry from the process table. In the case of a child whose parent has terminated, the process table is cleared upon exit and no zombie process is created. If accounting is enabled, **exit** writes an accounting record.

The **wait** system call suspends the calling process until a child process **exits**, a child stops in trace mode (the child is traced by its parent), or the caller receives a signal. A **wait** system call passes termination status to the parent process, 1 byte (high) passed by **exit** and 1 byte (low) of system status. This system call also removes zombies from the process table.

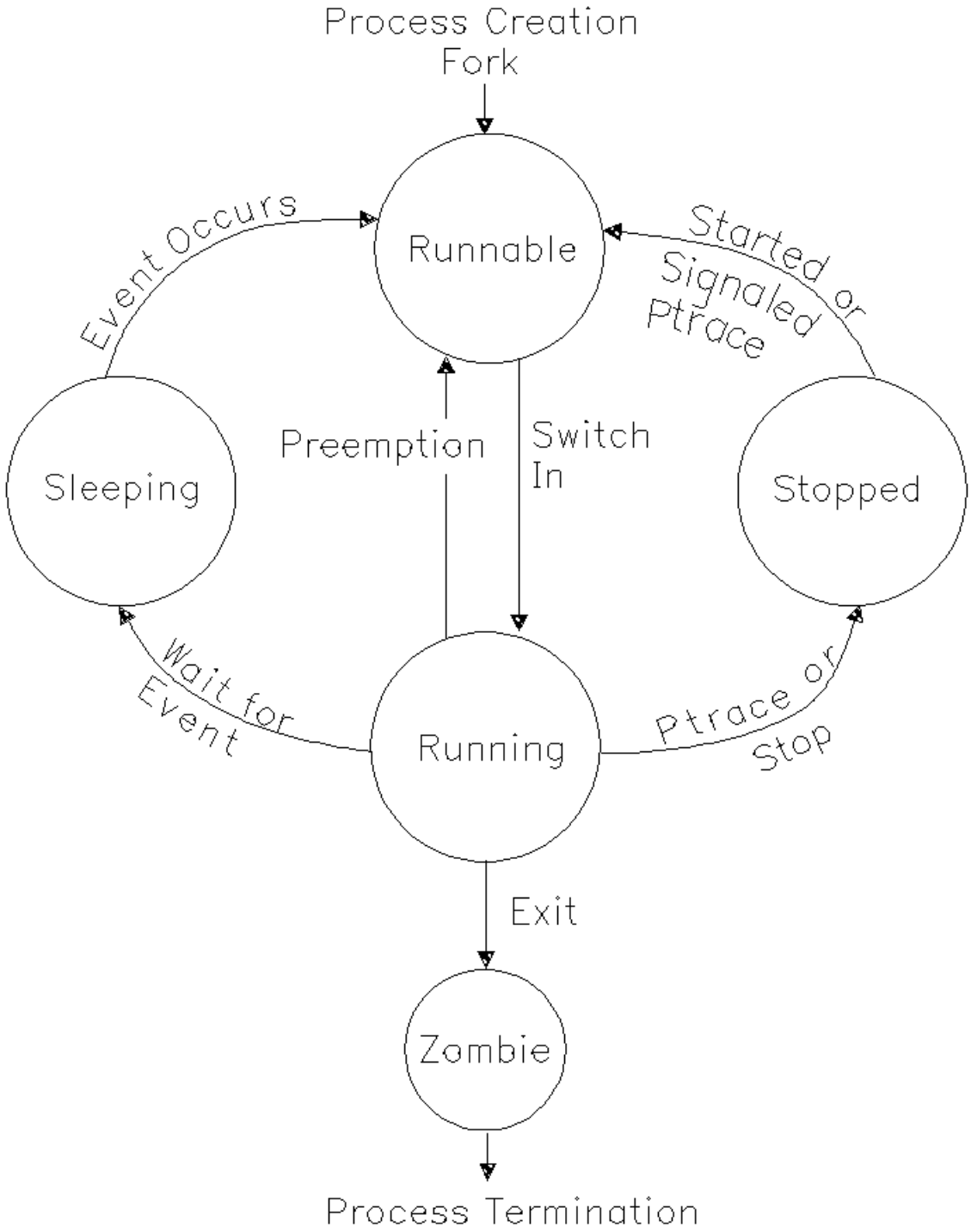
The following scenario discusses a parent process and child process relationship and the system calls to synchronize them.

A parent process executes a **fork** system call, producing a new process. The new process executes an **exec** system call creating a child process with a new identity. This is similar to the sequence **shell** uses when it runs a program. The **wait** system call causes a parent process to wait for the child to finish processing. When running interactively, the **shell** process executes a **fork** system call, the child process (shell running in the new process) executes an **exec** system call for the required program, and the parent process (shell) executes a **wait** system call to wait for the child to finish running. When the child executes an **exit** system call, the parent causes the process table entry for the child to be removed and prompts for another command. When running in the background, the **shell** process simply prints the process ID of the child and does not wait for the child process to terminate. See the following diagram for the relationship of the parent and child processes as described when they run interactively.



1.1.4.3.3 States of a Process

A process can be in one of many states. A process can be ready to run, running, sleeping (waiting on an event), stopped, or ended. The scheduler determines which order the competing processes execute. The following diagram shows the process states and the events that change the states.



Only one user process is active or running at any given time. All other user processes are suspended from running. For example, a process that is waiting for any of its child processes to end waits for an event that is the address of its own process table entry. When a process terminates, it signals the event represented by the process table entry of its parent.

When the event occurs, the process is awakened. When a process is awakened, it is ready to run, which means it is eligible to be dispatched. Normally, processes run to completion unless they sleep. They sleep for reasons such as waiting for input or output to complete, waiting for a locked resource to become available, waiting for an event to occur or signals from other processes. At each timer interrupt, the timer interrupt routine examines the process queues, and may cause a process switch. When a process is sleeping, it may be swapped out of memory. The process switch routine will not restart a process that is swapped out. It checks that kernel and user data for a process are addressable before it restarts the process.

A process that relinquishes control of the processor is usually waiting for some I/O to be performed. In that case, the process, while inside the kernel, issues a **sleep** call specifying **chan**, which is usually the address of the kernel data structure, and specifies a wakeup priority. It normally remains in a sleep state until a **wakeup** call is issued specifying the same **chan**. If the wakeup priority is low enough for the signal to be processed, the process is awakened and restarted in the same mode prior to sleep. Sometimes many processes may be waiting on the same event to occur, such as memory allocation. Since this is possible, when the process returns from sleep, it must first check that the event or resource was not seized by another process waiting on the same **chan**. If the resource is not available, the process issues another **sleep** call. A process may also be stopped, either because it is being traced (see "ptrace" in topic 1.2.212) or because it received a signal whose current action is to stop the process (see "sigaction, sigvec, signal" in topic 1.2.263).

1.1.4.4 Priority Computation

Each process has an assigned priority. User processes are assigned low priorities. The scheduler uses the process priorities to dispatch processes. It dynamically calculates process priorities to select the inactive (but ready-to-run) process to run when the currently active process stops. A kernel process that is sleeping always sleeps at a priority higher than any user process. System processes also have higher priority than any user process. Since a system process always runs in kernel mode, it is not able to be preempted.

User process priorities are assigned by an algorithm based on the ratio of the amount of compute time to real time recently used by the process. At every tick of the system timer, the **p_cpu** field (processor usage) in the process table for the running process is incremented. The compute time to real-time ratio is updated every second. Using negative exponential distribution, the kernel decreases **p_cpu** by half its value for every process at or above the base user level and recalculates the priority of the processes. Processes that accumulated a lot of execution time have less priority than processes with very little execution time. A user process can execute a **nice** system call to induce a bias in the calculation. Ordinary user processes can only decrease their priority, while root user processes can either increase or decrease their priority.

1.1.4.5 Signals

A ***signal*** is an event that interrupts the normal execution of a process. The set of signals is defined by the AIX system, and they are listed in the discussion of "sigaction, sigvec, signal" in topic 1.2.263. All signals have the same priority.

A process can specify a signal handler subroutine, which is to be called when a signal occurs. It can also specify that a signal is to be blocked or ignored or that a default action is to be taken by the system when a signal occurs.

A global signal mask defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent. It can be changed with a **sigprocmask** or **sigsuspend** system call. While a signal handler is executing for a given signal, the signal that caused it to be called is blocked, but other signals can occur. When the handler finishes, the signal is again unblocked.

Normally, signal handlers execute on the current stack of the process. This can be changed, on a per-signal basis, so that signal handlers execute on a special signal stack.

When a signal is sent to a process, it is added to a set of signals pending for the process. If the signal is not currently blocked, it is delivered to the process. When a signal is delivered, the following actions occur:

1. The current state of the process execution context is saved.

2. A new signal mask is calculated, which remains in effect for the duration of the process's signal handler or until a **sigprocmask** or **sigsuspend** system call is made. The new mask is formed by logically ORing the current signal mask, the signal being delivered, and the signal mask associated with the handler to be called.

3. If the signal handler is to execute on the signal stack, then the current stack is changed to the signal stack.

4. The signal handler is called. The parameters that are passed to the handler are defined in the following description.

   The signal-handler subroutine can be declared as follows:

   ```
   handler (sig, code, scp)
   int sig, code;
   struct sigcontext *scp;
   ```

   The **sig** parameter is the signal number. The **code** parameter provides additional information about certain signals. The **scp** parameter points to the **sigcontext** structure that is later used to restore the process's previous execution context. The **sigcontext** structure is defined in **signal.h**.

5. If the signal-handling routine returns normally, the previous context is restored and the process resumes at the point at which it was interrupted. The handler can cause the process to resume in a different context by calling the **longjmp** subroutine. (For information on how to save and restore the execution context, see "setjmp, longjmp, _setjmp, _longjmp" in topic 1.2.250.)

After a **fork** system call, the child process inherits all signals, the signal mask, and the signal stack from its parent.

The **exec** system calls reset all caught signals to the default action. Signals that cause the default action continue to do so. Ignored signals continue to be ignored, the signal mask remains the same, and the signal stack state is reset.

When the **longjmp** subroutine is called, the process leaves the signal stack if it is currently on it and restores the signal mask to the state when the corresponding **setjmp** call was made. See "sigaction, sigvec, signal" in topic 1.2.263 for enhanced signal information.

The operating system has five signal classes:

Hardware signals occur as the result of conditions such as arithmeti exceptions, illegal instruction execution, or memory protection violations.

Software signals are generally user-initiated interrupts Termination, quit, and kill are signal types that represent various levels of user or program-initiated signals to a process. In addition, timer expiration can be signaled with software-driven alarm signals.

A process can be notified of an event that occurred based on som descriptor, or nonblocking operation that completes. A process can also request a catastrophic condition signal.

Processes can be stopped, restarted, or can receive notification o state changes in a child process.

Processes can receive threshold warnings when the processing unit tim limit or a file size limit is reached.

The kernel also contains additions and modifications to enhance the unsolicited interrupt signal system for kernel-to-process communications.

1.1.5 File System Management

Files within the file system are grouped into directories, and the directories are organized into a hierarchy. At the top of the hierarchy is a directory called the root directory. This directory is designated as **/** (slash). The root directory contains some system-related files and standard directories such as **/bin, /usr, /dev, /etc,** and **/lib**. Files can be attached anywhere within the hierarchy of directories.

A file is a one-dimensional array of bytes uniquely identified by a global file system (***gfs***) number and an inode number (***i-number***). Data within the file is located in blocks. The logical blocks in the AIX file system are 4096 bytes long.

Subtopics

1.1.5.1 Types of Files

AIX file system files can be directory files, ordinary files, symbolic links, or special files. All files have read, write, and execute permissions for the owner, group, and others. The read, write and execute permissions on a file are granted to a process if one or more of the following are true:

The effective user ID of the process is 0. This user is said to hav superuser authority.

The effective user ID of the process matches the user ID of the owne of the file and the appropriate access bit of the **owner** portion (700 octal or 0x01c0) of the file mode is set.

The effective user ID of the process does not match the user ID of th owner of the file, and the effective group ID of the process or one of the group IDs in the concurrent group list of the process matches the group of the file and the appropriate access bit of the **group** portion (070 octal or 0x38) of the file mode is set.

The effective user ID of the process does not match the user ID of th owner of the file, and the effective group ID of the process does not match the group ID of the file, and the appropriate access bit of the **other** portion (007 octal or 0x07) of the file mode is set.

Otherwise, the corresponding permissions are denied. The group ID of a newly created file or directory is the effective group ID of the process unless the ISgroup ID is turned on in the parent directory. In this case, the file inherits the group ID of the directory.

In a directory, however, the read, write, and execute permissions are interpreted differently from ordinary files. Read permission for a directory indicates that standard utility programs are allowed to open and read the information in the directory. Write permission for a directory indicates that files in the directory can be created or removed. Execute permission for a directory indicates that a user can search the directory for a file name. Denying search privileges for a directory provides protection against using files in that directory. A request to change location into a directory where execute permission is denied cannot be performed.

AIX permits file names to be up to 255 characters long. It is possible for one file to have several names. Any printable character other than / (slash) can be used in the name. Names containing unprintable characters, space characters, tabs, and shell metacharacters are not recommended. The AIX file system reserves file names
. (dot) and **..** (dot dot); therefore these names cannot be used as file names.

Subtopics
1.1.5.1.1 Directory Files
1.1.5.1.2 Ordinary Files
1.1.5.1.3 Special files
1.1.5.1.4 Symbolic Links
1.1.5.1.5 Hidden Directories

1.1.5.1.1 Directory Files

The AIX file system hierarchy centers around directory files. Directory files contain lists of files. The AIX operating system maintains the directory files. Executing programs can read the directory files, but AIX prevents programs from directly changing directory files to protect the information in the directory files. Programs may add entries to directories by requesting the system to create a file. The system is responsible for making the changes to directory files. Files listed in a directory can be ordinary files, directory files, or special files.

Because the internal format of directories is complex and uses variable-length fields, it is best to access directories with the **opendir**, **readdir**, and related library routines (see "directory: opendir, readdir, telldir, seekdir, rewinddir, closedir" in topic 1.2.60). These routines parse directory contents into convenient structures for you. They are quite efficient and programs using them are portable among all computers using the AIX Operating System.

1.1.5.1.2 Ordinary Files

Ordinary files are attached to directories. An ordinary file might contain an executable program, document text, or other types of information that can be processed. There are two types of ordinary files: text files and binary files. Text files normally contain ASCII (American Standard Code for Information Interchange) characters. Binary files contain 256 possible values for each byte. Text files can easily be shared between an AIX/370 process and an AIX PS/2 process. Binary files, however, often require special programming to be shared between AIX/370 and AIX PS/2 processes. This is because of hardware differences in the way bytes are ordered within 16-bit and 32-bit numeric values. The System/370 architecture stores the first addressable byte of a 16-bit or 32-bit number as the high order byte, while the 80386 architecture stores the first addressable byte as the low order byte. AIX provides routines to assist applications in accessing binary files (see " pad: sflip, sflipa, lflip, lflipa, get_howflip, PAD, PADOPEN, PADCLOSE" in topic 1.2.200).

1.1.5.1.3 Special files

Special files are used to provide a convenient channel for accessing input and output (I/O) mechanisms to devices. For each I/O device, including memory, there is a special file. Most special files are found in the **/dev** directory. Special files provide an interface between application programs and the AIX kernel routines dealing with the devices. The names of the special files indicate the type of devices with which they are associated. Special files are read and written just like ordinary files, except read and write requests activate the associated device.

There are two types of special files: character and block. Some devices, such as a terminal, handle one character at a time. The character special files provide access to character I/O devices. Some I/O devices, such as a disk, transfer data in blocks at a time for efficiency. The block special files provide access to block I/O devices. A character special file can also be created for a block I/O device. This provides a ***raw interface*** to the device. The raw interface is usually more efficient than the block interface since data often can be copied directly between the process's address space and the device.

No characters are stored in a special file. When a directory that contains special files is listed, it identifies major and minor device numbers associated with the device rather than file length. The major device number identifies the type of I/O device that the file references. The minor device identifies the specific device when multiple devices of the same type exist, such as terminals.

A ***pipe*** is a special type of file used for simple one-way interprocess communication. It is created and opened with the **pipe** system call and operated on with the **read** and **write** system calls. A FIFO special file is similar to a pipe, except that it is named in the file system. It is created with the **mknod** or **mkfifo** system call and opened with the **open** system call. Data written to a pipe or FIFO (first-in first-out) special file is read on a first-in first-out basis. Facilities are provided to make **read** system calls wait until data is available in the pipe, and **write** system calls wait until data is removed from a full pipe.

Sockets are part of a more extensive interprocess communication facility. A socket is an endpoint for two-way communication between processes. Each socket has queues for sending and receiving data.

Sockets are typed according to their communication properties. These properties include whether messages sent and received at a socket require the name of the partner, whether communication is reliable, the format used in naming message recipients, and so on.

Each instance of the system supports some collection of socket types; consult "socket" in topic 1.2.275 for more information about the types available and their properties.

Each instance of the system supports some number of sets of communication protocols. Each protocol set supports addresses of a certain format. An Address Family is a set of addresses for a specific group of protocols. Each socket has an address chosen from the address family in which the socket was created.

1.1.5.1.4 Symbolic Links

A ***symbolic link*** is like a regular file, except it contains the name of another file or directory. When a symbolic link is encountered, the contents of the symbolic link is concatenated with the remaining portion of the path name to create a new path name. The system restarts the path name expansion with this new path name. If the contents of the symbolic link is a path name that begins with a slash (/), the name is expanded starting at the root directory; if not, the directory containing the symbolic link is the starting point for further name expansion.

When the system encounters a symbolic link whose first component is the string **<LOCAL>/**, the system substitutes the current value of the process's **<LOCAL>** alias (see "getlocal, setlocal" in topic 1.2.102) for that component and continues the path name evaluation. The directories **/tmp** and **/dev** are normally symbolic links which use the **<LOCAL>** alias. In this way each machine in a TCF cluster can have a separate **/tmp** and **/dev**. Other uses are similar, like using them for the system administration files. In this way compatibility of the application is maximized and at the same time availability is maximized as the files can be maintained on the site to which they apply. The **<LOCAL>** alias can vary from process to process so the use of this alias with symbolic links creates a path name that in different processes may result in a given path name's naming different objects.

1.1.5.1.5 Hidden Directories

A ***hidden directory*** is a regular directory which is treated specially during path name expansion. A hidden directory is constructed by making a regular directory and then issuing the **chhidden** system call with the regular directory as an argument.

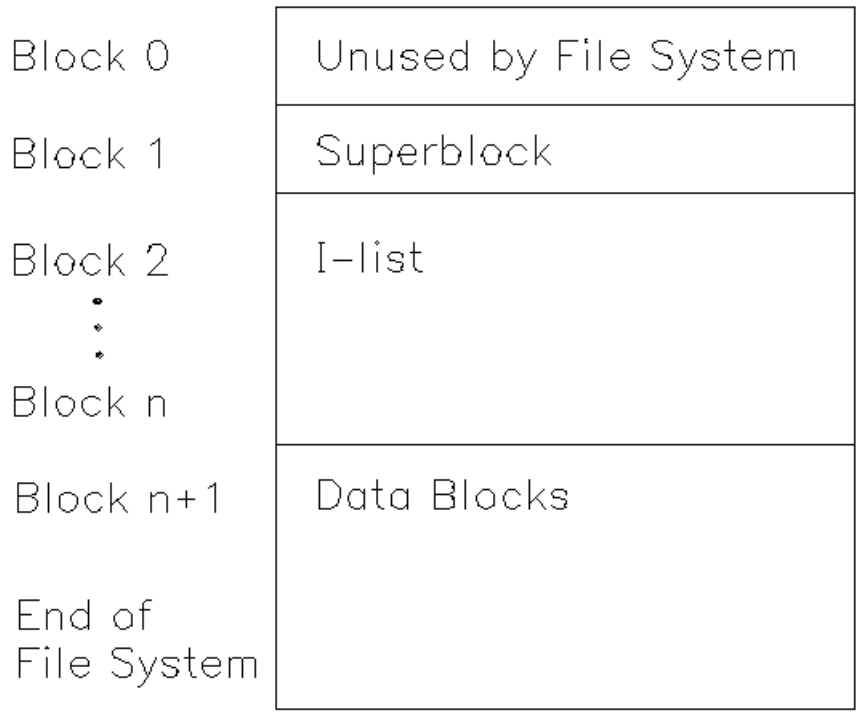
Normally, when a hidden directory is encountered during the evaluation of a path name, the system uses the process's site path (see "getspath, setspath" in topic 1.2.122) to automatically select a component within the hidden directory. This evaluation operates as follows: for each element in the site path, starting with the first, the system determines whether the specified site, or a site of the specified type, is available. If so, the hidden directory is searched for the component name of the selected machine type. The first time this procedure results in a match to a file within the hidden directory, that file is selected and the evaluation completes.

Only regular and special files are permitted in hidden directories. In particular, this excludes from hidden directories symbolic links and directories (regular or hidden).

Some system calls are defined to disable the hidden directory mechanism when their path names are evaluated, thus treating a hidden directory as though it were a regular directory. In addition, the character @ may be placed at the end of a file name to disable the hidden directory mechanism for that file. All system calls will remove the final character from a path name component if that final character is an @, regardless of whether the file name names a hidden directory. One must use @@ at the end of a path name component to name a file which actually does end with @.

1.1.5.2 File System Layout

For this discussion, the device that contains the file system is a minidisk with logical data blocks of 4096 bytes. Therefore, a unit of disk storage or block is 4096 bytes. Blocks are numbered sequentially from the beginning of the minidisk, starting with 0. A file system is logically separated into four sections as shown in the following figure.



1.1.5.3 Block 0

The file system does not use block 0. This block usually contains system bootstrap information.

1.1.5.4 Super block

Block 1 is the ***super block***. See "fs" in topic 2.3.20 for a detailed description of the contents of this structure. This block is used to keep track of the file system. Some of the file system information contained in the super block is:

File system size in logical block
File system nam
Number of blocks reserved for inode
The inode lis
The free-block lis
The global file system (gfs) number

1.1.5.5 I-list

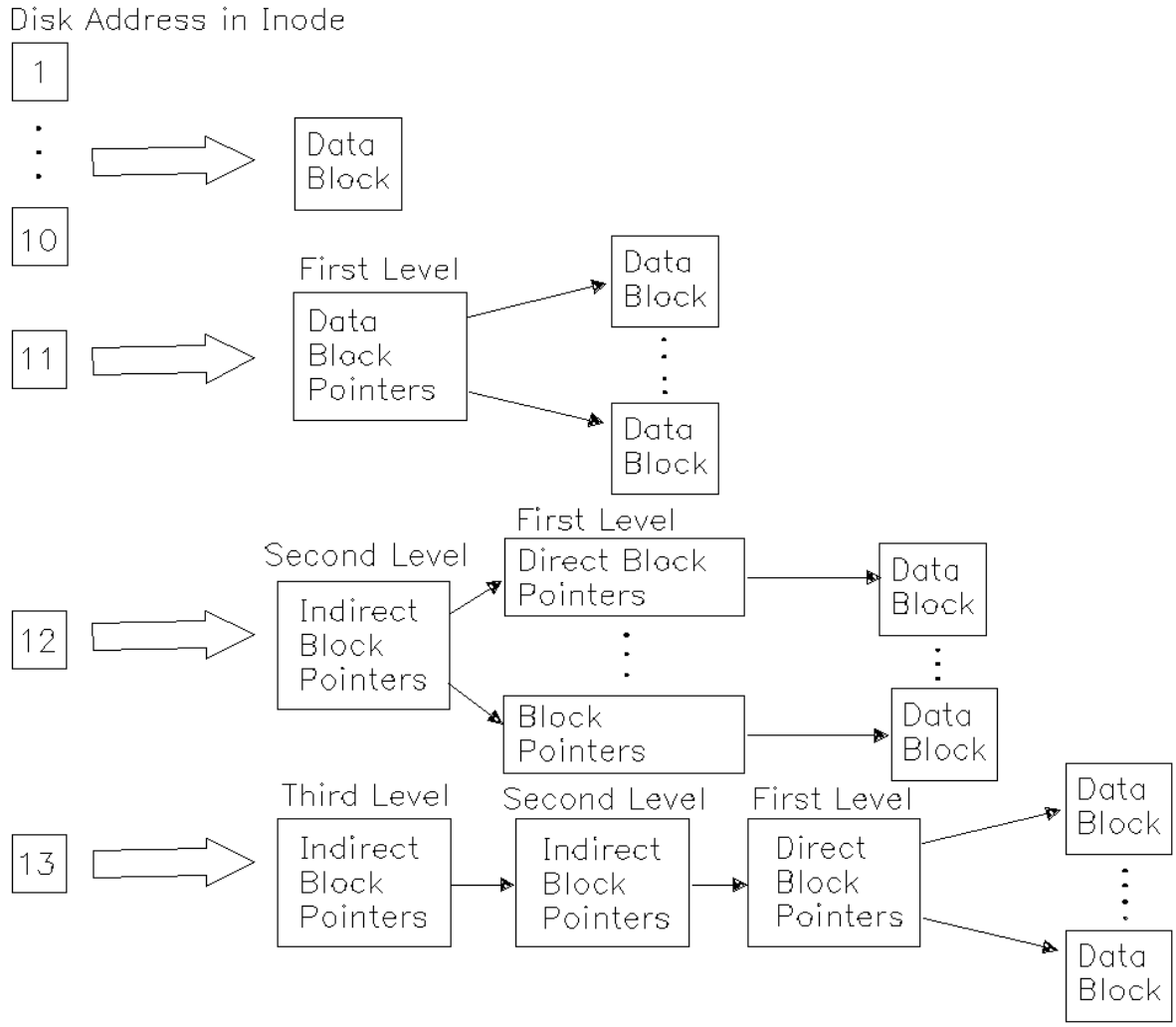
Blocks 2 through **n** are the ***i-list***, which contains structures relating a file to the data blocks on disk. The size of this section depends on the size of the mounted file system. Each structure, called an ***inode***, is 512 bytes long. Each inode designates a file. See "inode" in topic 2.3.29 for the detailed content of the inode structure of an ordinary file or directory. Each inode structure is sequentially numbered from 1 to a maximum number, as defined in the super block. Each index number, or ***i-number***, designates a 512-byte inode and is used as an offset within the i-list. I-number 1, the first 512 bytes, is not allocated by the file system. I-number 2 is the inode of the root directory. The remaining i-numbers are allocated by the file system. If a file is less than 384 bytes long, the data of the file is stored in the inode. The inode contains information about each file such as:

- Mode and type of fil
- Length of fil
- ID numbers of owner and grou
- Relevant dates and time
- Number of link
- Location of file block
- Data for small file
- Number of blocks allocated

Subtopics
1.1.5.5.1 Inode Addresses

1.1.5.5.1 Inode Addresses

An inode contains thirteen 4-byte disk addresses. The following figure shows the use of the disk addresses in the inode.



Addresses 1 through 10 point directly to the first 10 disk blocks in the file. Addresses in indirect blocks are 4 bytes long. If the file is larger than 10 blocks, address 11 points to a first level indirect block containing the next blocksize÷4 block addresses in the file. This is called indirect addressing. An indirect block contains 4096÷4 or 1024 addresses. A larger file requires use of the address 12. This address points to a second-level indirect block, which contains addresses of up to blocksize ÷ 4 first-level indirect blocks. If a file could be larger, address 13 would be required. This address would point to a third-level indirect block, which would contain the addresses of up to blocksize ÷ 4 second-level indirect blocks. (Since AIX uses 4096-byte blocks and file offsets must be specified by a 32-bit number, it is not possible to create a file which requires triple indirection.) Any of these addresses can be 0, indicating holes in the file, which are read as binary zeros. Indirect block numbers can be 0 when the file contains large holes.

1.1.5.6 I-number Allocation

The file system tracks free i-numbers that are available. It maintains a list of inodes available for allocation in the super block. The super block contains the following information to allocate free i-numbers.

s_inode, an array containing the next free i-numbers to be allocated to files.

s_ninode, the count of free i-numbers in the array. This is used as an index into the **s_inode** array.

s_tinode, the total number of inodes in the file system.

Allocating an i-number to a file when **s_ninode** is greater than 0, **s_ninode** is decremented to get the next available i-number from **s_inode**. If **s_inode[s_ninode**] is 0, the next free i-numbers available from the i-list are placed onto the array and another attempt to allocate is made. Freeing an i-number when **s_ninode[s_ninode**] is less than maximum, places the freed i-number into the array and the count increments.

1.1.5.7 Data Blocks

The last section of the file system consists of ***data blocks***, which contain data stored in files, indirect blocks that point to other data or indirect blocks for large files, or blocks that are available for data. These blocks are 4096 bytes long. The inode contains the addresses of the data blocks that are already used in files. Otherwise, the data blocks are free and available for allocation to a file.

1.1.5.8 Free-block List

The file system maintains a list of all free blocks in a ***free-block list***. The free-block list is a linked list of pointer blocks. A free block is a block that is not allocated to the super block, inodes, indirect blocks, or files. Blocks are allocated dynamically to a file when needed from the data block section of the file system. In order to track data-block allocation, the super block contains the following:

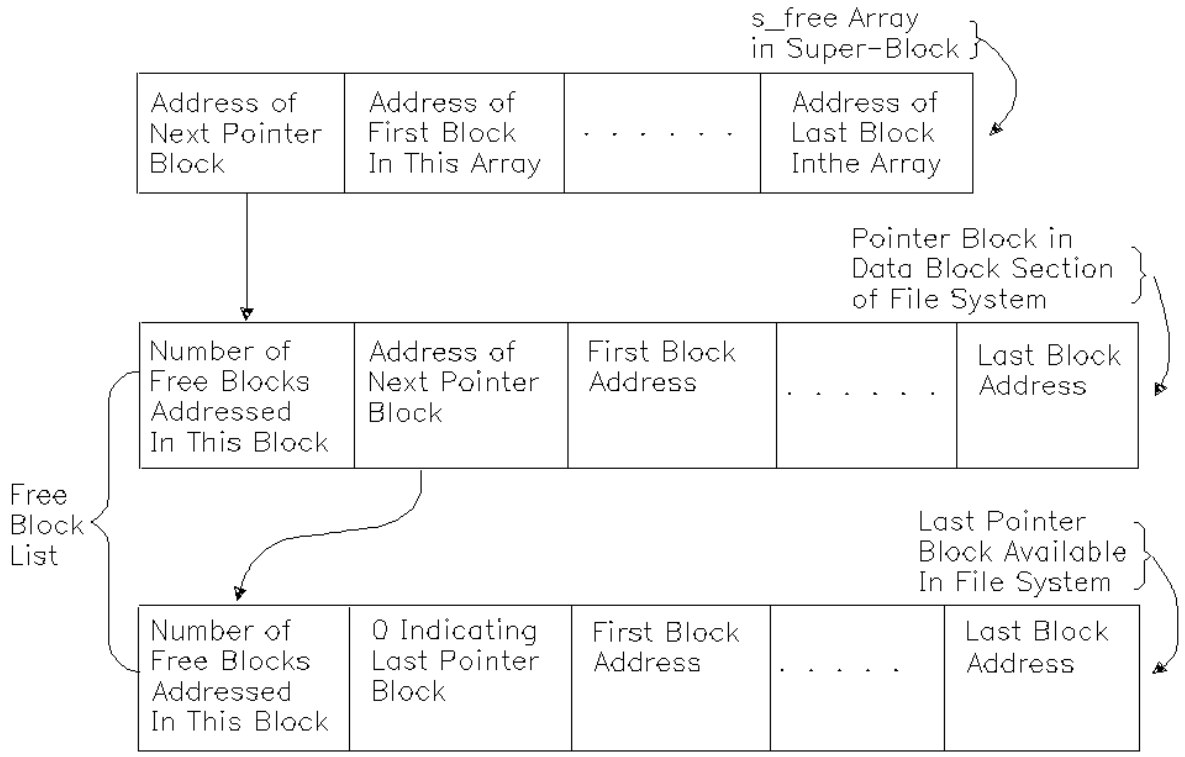
s_free, an array of free block addresses.

s_nfree, the number of free blocks in the **s_free** array. This is used as an index into the **s_free** array.

s_tfree, the total number of free blocks available in the file system.

1.1.5.9 Allocating Blocks

Each pointer block in the free-block list contains a count of the number of entries in the block, up to 600, and the address of the next pointer block. If the pointer has a value of 0, this indicates the last pointer block in the file system. The first long integer in each pointer block is the number, up to 600, of free blocks addressed in the block. The next long integer is the address of the next pointer block available. The next 599 long integers contain the addresses of 599 free blocks. The following figure shows the relationship of the free-block list and **s_free** array.



The file system allocates free blocks using the **s_free** array of pointers in the super block. The pointer block information is copied into the **s_free** array in the super block as follows. **s_free[0]** contains the address of the next pointer block in the free-block list. The remainder of the **s_free** array contains addresses of the free blocks in this pointer block. **s_free[s_nfree-1]** contains the address of the next free block available to be allocated.

Allocating a block causes **s_nfree** to be decremented to locate the next available block. If decrementing **s_nfree** caused its value to become 0, this indicates that more blocks are not available in the **s_free** array. Therefore, the address found is the location of the next pointer block. File system management reads the pointer block into the super block, placing its first long integer in **s_nfree** and copying the next 600 long integers, which are addresses, into the **s_free** array. If the location of the next free block is 0, indicating the end of the chain, then new blocks are not available in the file system. This indicates a file system out-of-space error condition.

When a block is freed and **s_nfree** has a value less than 600, the new block

address is added to the **s_free** array and **s_nfree** is incremented. When a block is freed and **s_nfree** equals 600, **s_nfree** and the **s_free** array are copied into the freed block and the value **s_nfree** becomes 0. The file system manager updates the **s_free** array as previously described when **s_nfree** was equal to 0.

When changes are made to an existing file, AIX uses a ***shadow page*** mechanism which means that changes are written to newly allocated blocks rather than to the existing blocks of a file. This mechanism allows the disk version of a file to remain intact in the unlikely event of a system crash or power failure. It also enables the atomic file commit mechanism (see "fsync, fcommit" in topic 1.2.87).

Subtopics
1.1.5.9.1 Directory Contents

1.1.5.9.1 Directory Contents

A ***directory file*** is like an ordinary file except that it cannot be written by a user. See "dir" in topic 2.3.16 for a detailed description of a directory. A bit in the inode identifies the file as a directory file. The directory contains an entry for each file or subdirectory within it. A directory block contains one or more directory entries. A directory entry is a variable length record described by the **dirent** structure. Each directory entry is between 16 and 1024 bytes in length and either describes a file whose name length is between 1 and 255 bytes, or is an unused entry. The first four bytes of each directory entry are the i-number for the file. The next two bytes hold the length of the entire entry (rounded up to the nearest multiple of 16). The next two bytes hold the actual length of the file name (not including NULL padding). The remaining bytes of the directory entry hold the null-terminated file name.

For compatibility with other systems, programs which read directories with the **read** system call are presented with directory entries in the old, 16-byte directory entry format. To see the true contents of a directory, a program must use the **readdir** or **readx** routines.

1.1.5.10 Path Name Resolution

A path name through the file system is a route of directories and inodes. A direct path starts at the file system root. A relative path starts at the current directory. The last name in the path references a file. The following example shows the resolution of a direct path; the resolution of a relative path is similar.

Subtopics
1.1.5.10.1 Full Path
1.1.5.10.2 Relative Path
1.1.5.10.3 File System Data Structures

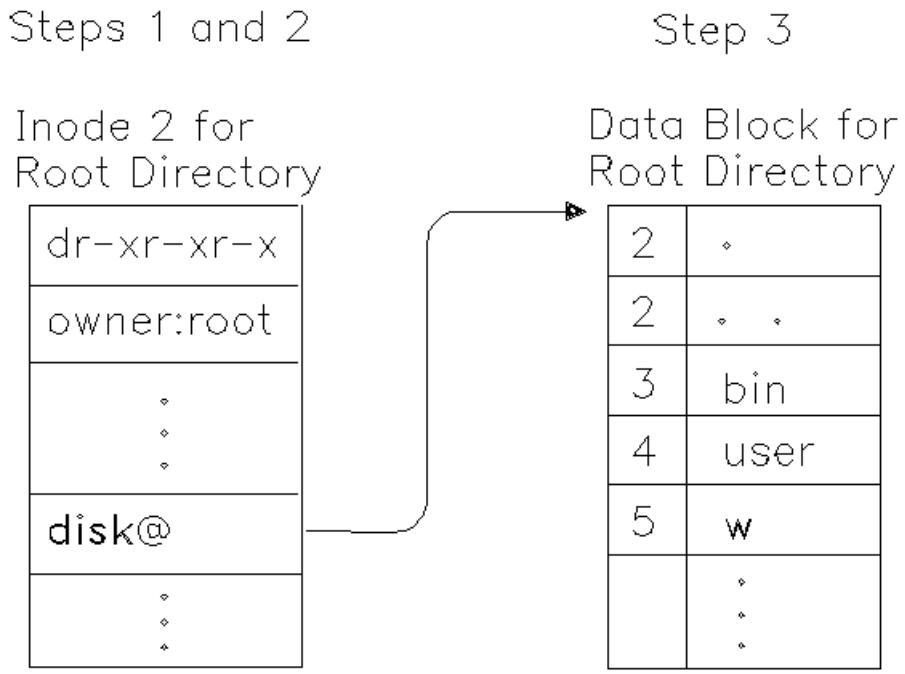
1.1.5.10.1 Full Path

The following describes accessing a file with a full path name. Consider the path name **/w/z**. This path starts at the root directory. It leads from the root directory to the directory **w** and then to the file **z**.

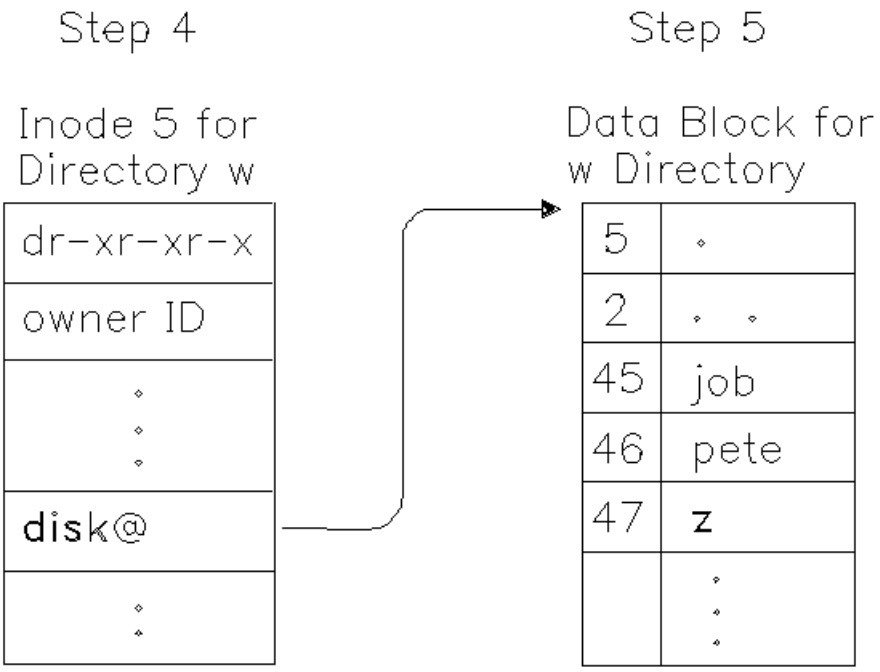
1. Read address 1 of i-number 2 (root) for the address of the root directory.

2. Read the inode for the root directory.

3. Use the information in the root directory to search the root directory for the name **w** and its i-number.

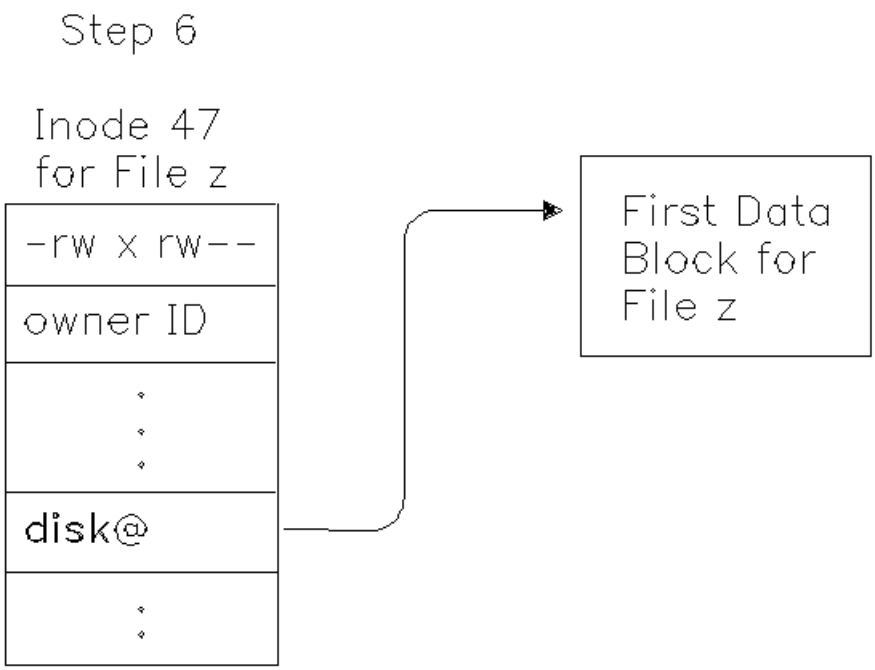


4. Read the i-number for **w**.



5. Use the information in the **w** inode to search the **w** directory for the file named **z** and its i-number.

6. Read the inode for **z**. The addresses for the file blocks assigned to the files start at **di_addr** within the inode. The first 10 are direct addresses as previously described; the last three are indirect.



1.1.5.10.2 Relative Path

Consider the relative path name **..(dot dot)/x/y**. The path leads from the current directory, to the parent of the current directory, to the parent subdirectory **x**, and finally to the file named **y** in the directory **x**. In order to follow this path, the system performs the following steps:

1. Read the inode of the current directory.

2. Use the information in the inode for the current directory to search the current directory for the name **..** (dot dot) and its i-number.

3. Read the i-number for **..** (dot dot).

4. Use the information in the **..** (dot dot) inode to search the parent directory for the file named **x** and its i-number.

5. Read the inode for **x**.

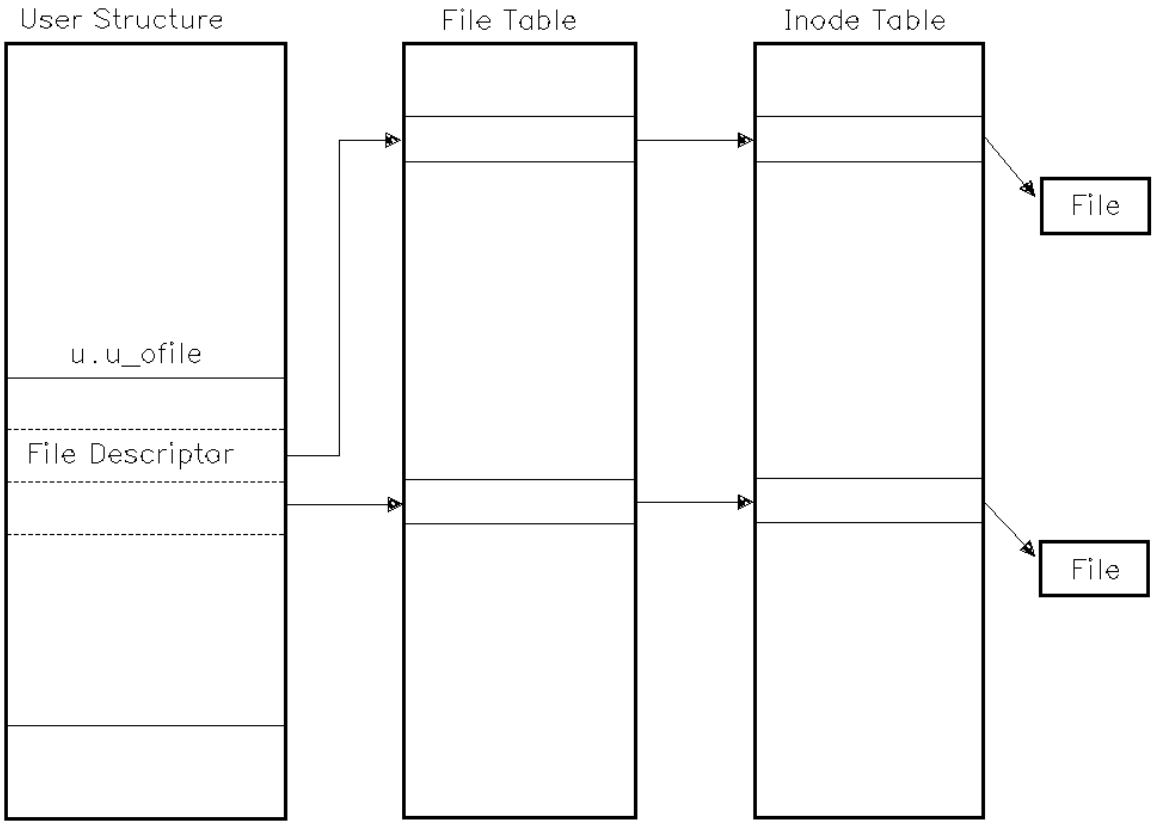
6. Use the information in the **x** inode to search the **x** directory for the file named **y** and its i-number.

7. Read the inode for **y**. The addresses for the file blocks assigned to this file start at **di_addr** within the inode. The first 10 are direct addresses as previously described; the last three are indirect.

1.1.5.10.3 File System Data Structures

The kernel maintains structures in memory, along with the super block, to access files in the file system. These structures include the inode table and file table.

Access to the files in the system begins at the per-process data region in a process (user structure). System calls provide access to file system services for user processes. The most common functions performed are **open, create, read, write, lseek,** and **close**. The user structure contains an array, **u.u_ofile**, that is indexed by file descriptor values. This array of entries contains the addresses of the file table entries for each file opened or created by the process and used for I/O operations. Descendants of the process inherit the contents of the **u.u_ofile** array. The following figure shows the data structure relationships for accessing two files. The format of the user structure is found in **/usr/include/sys/user.h**.



All operations on files are performed with the aid of the corresponding inode table entry. When the system accesses a file, it locates the corresponding inode (see "Path Name Resolution" in topic 1.1.5.10), allocates an entry in the inode table and reads the inode into memory. The entry in the inode table is the current version of the inode and is the focus of file system activity. The structure of an inode entry is found in **/usr/include/sys/inode.h**. The inode table contains the key information for accessing a file including flags, owner, mode, mounted-on device, i-number, and location of file blocks.

Another table the kernel maintains in memory for accessing files is the ***file table***. The structure of a file table entry is found in the

/usr/include/sys/file.h file in the file system. A file table entry is associated with **open** and **creat** calls for each file. Each entry in the file table contains the read/write offset of the file and a pointer to an inode. The user process maintains a file table entry for each file it opened or created. After a fork, the two processes share the file table entries. A separate open of a file that is already open shares the inode table entry but has distinct file table entries.

1.1.6 I/O Control

The kernel and user processes use calls to the system to access the I/O subsystem. System calls that perform I/O usually cause the calling process to be suspended (it relinquishes control of the hardware processor) while the I/O is being performed. Another process that is ready to run is dispatched.

The AIX kernel sends requests to the hardware devices using I/O device commands.

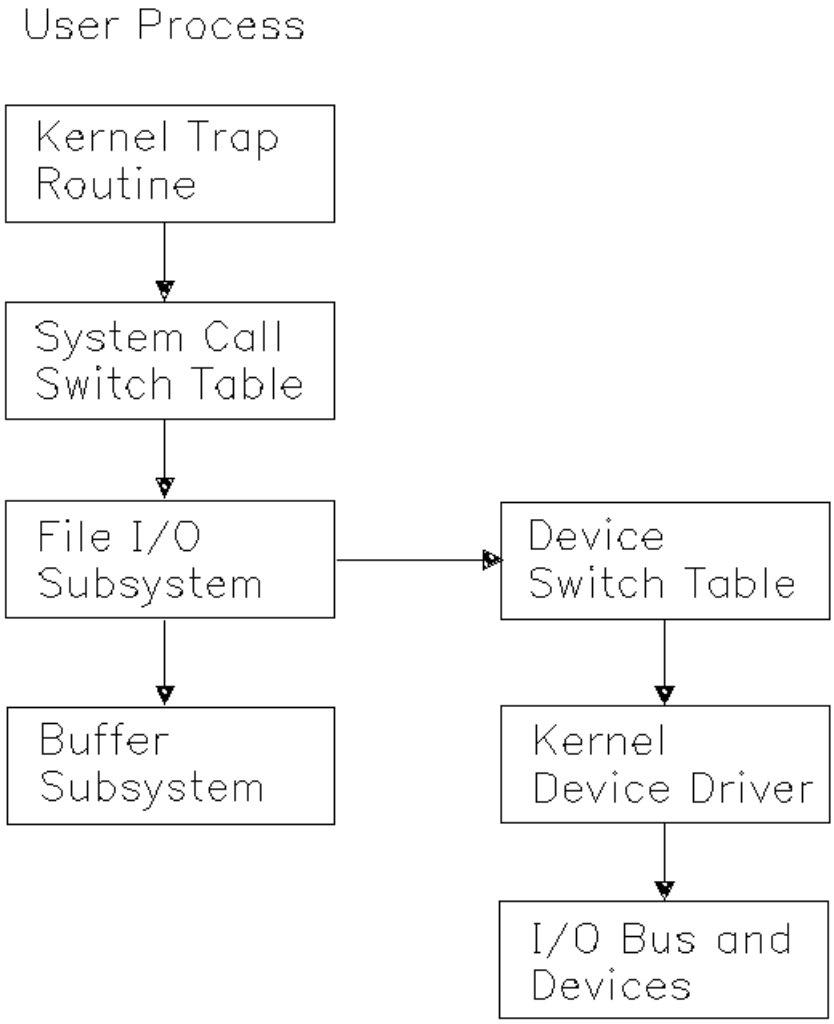
Each physical device that is attached to the system must communicate with the AIX kernel via a device driver. The AIX device drivers deal directly with actual device interrupts and directly place requests on the hardware bus.

Each AIX device driver is activated by a basic I/O system call (**open**, **read**, **write**, **ioctl**, **close**, and so on) from the application level.

The AIX device driver routine for the particular device class performs the requested I/O function to a device command. There are two basic device classes, one for block-oriented devices, and one for character-oriented devices.

Each AIX device driver is addressed by individual applications via a system call interface. The application builds device-dependent commands and data streams, and invokes the appropriate AIX device driver. To accomplish the system call, the device driver maps the system call inputs into commands to the device.

The AIX device drivers perform basic device error determination by reading device status to determine exception conditions. The following illustration shows an overview of the relationship in the control flow of the I/O subsystem.



Subtopics
1.1.6.1 Kernel Trap Routine
1.1.6.2 System Call Switch Table
1.1.6.3 File I/O Subsystem
1.1.6.4 Buffer Subsystem
1.1.6.5 Device Switch Table
1.1.6.6 Kernel Device Driver
1.1.6.7 Common Routines
1.1.6.8 I/O Data Structures
1.1.6.9 Device Management
1.1.6.10 Requests for Device I/O

1.1.6.1 Kernel Trap Routine

Each system call is interpreted as a request to perform a predetermined function. The function to be performed is determined by a trap handler in the kernel. This ***kernel trap routine*** is called in other instances besides system call handling. This routine also runs in cases of error conditions or interrupt handling.

During a system call, any error indicators are reset and the process return status is saved. Next, the system call is used to determine a system call number. (An integer value is assigned for each type of system call.)

1.1.6.2 System Call Switch Table

The system call number is used as an index into the ***system call switch table***. This table contains the address for the specific handler routine that handles the call. A call is made to the system call handler routine, which receives the parameters supplied by the user program along with the system call. This routine copies the parameters out of the user part of the process to the kernel part of the process.

1.1.6.3 File I/O Subsystem

The system call switch table contains many entry points into the ***file I/O subsystem***. Common entry points used are **open, close, read, write, lseek** and **ioctl** system calls. The file I/O subsystem determines whether the system call is to gain access to an ordinary file, a block special file, or a character special file. In the case of special files, this subsystem translates the file name into a major and minor number, which is used to select the device and/or routine.

1.1.6.4 Buffer Subsystem

The ***buffer subsystem*** maintains a system buffer pool that is used by block devices to read and write data. Requests for blocks found in the pool are returned immediately to the requester. If blocks are not found in the pool, the least recently used (LRU) buffer is freed and allocated.

1.1.6.5 Device Switch Table

The ***device switch table*** is used as an interface to the device drivers. The device driver ***major*** number is used to select the proper routine. The ***minor*** number selects one of multiple subdevices. See "Device Management" in topic 1.1.6.9 for more details about device drivers.

1.1.6.6 Kernel Device Driver

The device driver in the kernel issues I/O directly to the device. When the device driver has completed its task, it returns back to the process.

When the device routine returns, the return status is copied back into the user part of the process and the process resumes running. Some rescheduling of processes can occur upon return from kernel mode due to interrupts or errors while processing the system call. Execution starts in the program immediately after the system call unless an error occurs.

1.1.6.7 Common Routines

Kernel and user processes use calls to kernel routines as an interface to the I/O subsystem. These routines must prepare the system internal tables in order to ensure proper performance. These routines are invoked using system calls. The following describes the common routines used and their effects on tables maintained by the operating system.

Subtopics
1.1.6.7.1 Creat and Open
1.1.6.7.2 Close
1.1.6.7.3 Read and Write

1.1.6.7.1 Creat and Open

The **creat** and **open** routines create and/or open a file for reading or writing and return a file descriptor for the opened file. First, the file system directory is scanned to locate the named file. An inode is created if not found and an entry is placed in an ***inode table***. This entry is somewhat different from the inode as it exists on the disk. It contains a count of the users (used by **close**) and other incore-only information. There is one inode table entry for a given file. An inode table entry exists for an open file, the current directory of a process, or a directory file over which a file system is mounted.

For each open file, an entry in the array **u.u_ofile** exists in the user structure. The **read**, **write** or any other routines that perform operations on the opened file use the file descriptor returned as an index into this array. Array entries are pointers to corresponding entries in the **file** table that is maintained by the system.

Each **creat** or **open** of a file causes one file table entry to be created. If a file is opened by more than one process, this table contains multiple entries. After a process performs a **fork** system call, the resulting processes share the same entry of the opened file in the table. The **fork** system call increments the reference count entry in the table. This count is used by **close** to determine when the entry can be removed from the table. Additionally, it contains a pointer to an inode.

1.1.6.7.2 Close

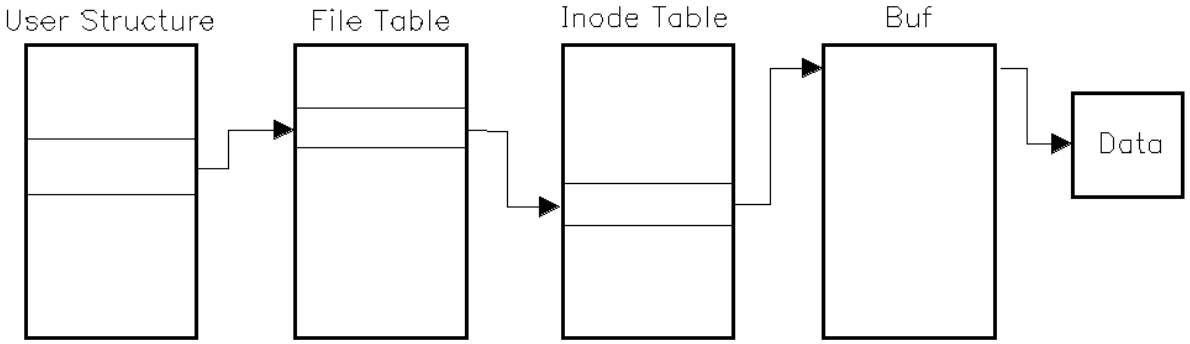
The **close** routine is called each time a process closes a file. When the last process closes the file, the inode table entry is removed. In some instances, buffers containing data for the file that are queued but not written, are written to the file before the **close** completes.

1.1.6.7.3 Read and Write

The **read** and **write** routines use parameters supplied by the user and the file table entry to set the variables **u.u_base, u.u_count,** and **u.u_offset** (in the user structure). These variables contain the user address of the I/O target area, the byte count for the transfer, and the current location within the file. It may be necessary to transform the current location into a logical block number or physical block number depending on the target.

1.1.6.8 I/O Data Structures

The operating system maintains data structures to track I/O processing to and from devices. The following figure shows these data structures and their relative relationship.



When an **open** or **creat** occurs, an entry is made in the file table. This table is referenced by pointers from the user structure using file reference numbers passed to system calls.

Another data structure is the ***inode table***, which contains one entry for each active inode. Each entry maintains an open count and a link count, which is used by **close**. This table is referenced by device number and i-number. The entry in this table is created by the **open** routine and removed by the **close** routine (when the open count and link count are 0). The inode table array of a file is found by following a series of pointers. The first pointer is in the user structure, which points to a file table entry, which points to the inode table entry. (See above figure.)

The ***user structure*** contains information accessed by the user process, kernel, and the device driver routines to perform device I/O requests. The elements of this block are needed when performing I/O.

Buf is a table of buffer headers maintained by the kernel and used for data read from or written to block devices. Each buffer header has

Flags (to show status information

Device and block-number fields (to identify which disk block has bee read into this buffer, or which disk block is to be written out from this buffer)

The pointer to the attached file system buffe

Forward and backwards pointers (to maintain two doubly linked lists the **b** list, which links buffers on one of several buffer hash chains to speed buffer lookup, and the **av** list which links buffers that are available for reuse).

The structure of the buffer header can be found in the file **/usr/include/sys/buf.h** in the file system.

1.1.6.9 Device Management

The operating system uses ***special files***, sometimes called ***device files***, to refer to specific hardware devices and device drivers. Special files, at first glance, appear to be files just like any other. They have path names that appear in a directory, and they have the same access protection as ordinary files. They can be used in almost every way that ordinary files can be used. However, an ordinary file is a logical grouping of data recorded on disk, but a special file corresponds to a device (such as a line printer), a logical subdevice (such as a large section of disk drive), or a pseudo-device (such as the physical memory of the computer, **/dev/mem**, or the null file, **/dev/null**). By convention, all special files supplied with AIX are located in the **/dev** directory.

Subtopics
1.1.6.9.1 Device Drivers
1.1.6.9.2 Major Device Number
1.1.6.9.3 Minor Device Number

1.1.6.9.1 Device Drivers

A ***device driver*** is a set of routines that are installed as part of the AIX kernel to control the transmission of data to and from a device. The major interface between the kernel and the device drivers is through the device switch table.

1.1.6.9.2 Major Device Number

A ***major device number*** designates which device driver in the operating system is to handle I/O requests. The major device number for each device is assigned in the **/etc/master** file, which is used in system configuration (see the **config** command in *AIX Operating System Commands Reference*).

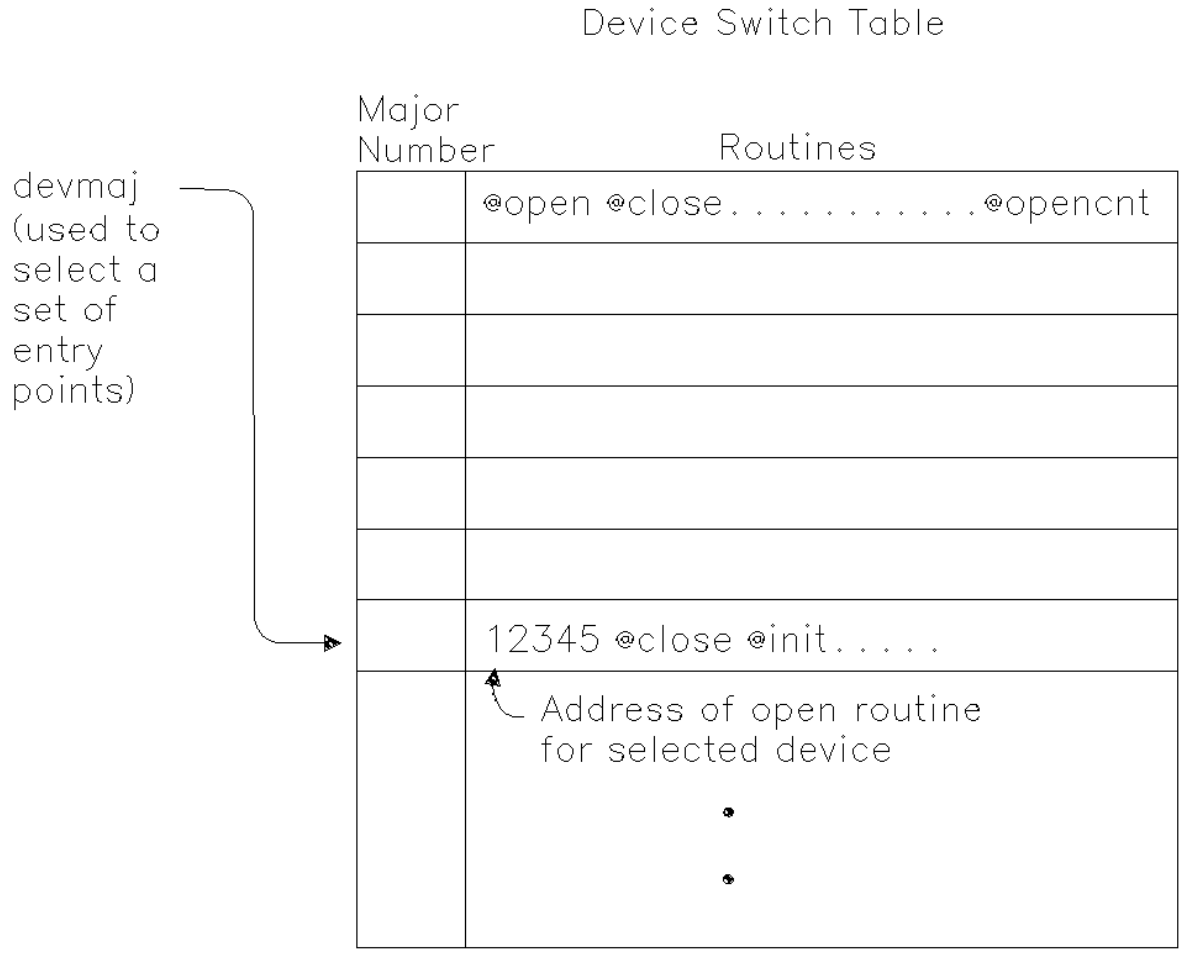
1.1.6.9.3 Minor Device Number

The interpretation of the ***minor device number*** is entirely dependent on the particular device driver. The minor device number is frequently used to index an array that contains information about each of several virtual devices or subdevices.

1.1.6.10 Requests for Device I/O

The operating system controls the processing of all user I/O requests and device interrupts. When a user program requests I/O to a device using system calls, control is transferred to the kernel. If the system call is to a device (not an ordinary file), the path pointer points to a special file. Special files describe the device and indicate to the system that the call is for a device. If the requested file is a special file, the system records the major and minor device numbers in the inode table entry.

All devices attached to the system are controlled by device drivers. The device drivers contain routines that specify the functions that can be performed by a device, such as read, write, open, and close. Each device has a set of driver routines that can be accessed by the kernel via a device switch table. The kernel uses the major device number designated in a corresponding special file as an index into the device switch table as shown in the next diagram. The minor number, which is passed as a parameter, selects one of a class of devices (such as a diskette drive) from a group of devices or specifies device characteristics.



After the kernel creates the inode table entry for the device, all references to the device use the inode number assigned to that device until the device is closed.

The following describes an overview of the processing of an I/O request to a device. When a call is made to a device, the kernel first runs the device-independent routines needed for the I/O request. Then, it determines the proper device driver routine to invoke for the required device-dependent process using the major number. Next, it calls the appropriate device driver routine. The requested I/O function is performed, control returns to the kernel. The kernel finishes processing the I/O request and returns control and any values to the user program.

When a device signals an interrupt to the processor (indicating I/O request completed), control is transferred to the interrupt vector in low memory. The interrupt vector first transfers control to the interrupt handler, which performs device-independent interrupt processing. Next, the device-dependent interrupt handler, which is part of the device driver software, is invoked. The interrupt handler processes the interrupt and returns control to the kernel. The kernel returns control to the process that had control of the processor at the time of the interrupt.

1.2 Chapter 2. System Calls and Subroutines

Subtopics

1.2.1 About This Chapter

This chapter gives detailed information about each of the system calls that are available in the AIX Operating System and the subroutines (also called ***functions***) that are available in standard AIX subroutine libraries. System calls provide controlled access to the operating system kernel.

The programming interface to the system calls is identical to that of subroutines. Thus, as far as a C language program is concerned, a system call is merely a subroutine call. The real difference between a system call and a subroutine is the type of operation it performs. When a program invokes a system call, a mode switch takes place so that the called routine has access to the operating system kernel's delicate information. The routine then operates in kernel mode to perform a task on behalf of the program. In this way, access to the delicate system information is restricted to a pre-defined set of routines whose actions can be controlled.

The operations performed by system calls are frequently more basic or "primitive" than those of subroutines. Many subroutines use system calls to perform more complex tasks. For example, the **open**, **close**, **read**, and **write** system calls perform very simple I/O operations; but many programs use a standard set of I/O subroutines that add data buffering to the I/O performed by the system calls. (See " stdio" in topic 1.2.283 for details about the Standard I/O Package.)

When an error occurs, most system calls return a value of -1 and set an external variable named **errno** to identify the error. The **errno.h** header file declares the **errno** variable and defines a constant for each of the possible error conditions. A complete listing of these error codes and their meanings can be found in Appendix A, "Error Codes." The specific meanings of the error codes that apply to each system call are listed in the "Error Conditions" section of each system call entry.

For an explanation of the "Syntax" section of each entry, see "Syntax" in topic 1.2.4. For an explanation of header files, see "Header Files" in topic 1.2.5.

1.2.2 System Calls

The following discussion is divided into sections that discuss groups of system calls that perform various operations.

Subtopics

1.2.2.1 Input/Output

The following system calls perform the basic input/output for all types of devices:

access	Determines whether the process has permission to access a file.
close	Closes a file associated with a file descriptor.
closex	**close** associated also with a character device driver.
creat	Creates a new file or replaces an existing file with an empty one.
dup	Duplicates an open file descriptor.
fabort	Cancels file changes.
fclear	Clears space in a file, freeing unused disk space.
fcommit	Commits file changes.
fsync	Forces changes to a file to be written to the disk.
ftruncate	Shortens a file.
ioctl	Controls I/O devices.
ioctlx	**ioctl** with additional device specific information.
lockf	Locks a region of a file from access by other processes.
lseek	Moves the read/write pointer of a file.
open	Opens a file or device for reading or writing.
openx	**open** with additional status provided.
pipe	Creates a pipe.
read	Reads data from a file, directory, socket, or device.
readx	**read** with communication with character device drivers.
select	Waits for an input/output to occur.
truncate	Makes a file shorter.
write	Writes data to a file, socket, or device.
writex	**write** with communication with char device drivers.

1.2.2.2 File Maintenance

The file maintenance calls change the access permissions of files, create directories, mount file systems, and perform a variety of other operations:

chdir	Changes the current directory.
chhidden	Changes the hidden attributes of a directory.
chmod	Changes the access permission mode of a file.
chown	Changes the owner and group IDs of files associated with a path name.
chroot	Changes the directory considered to be the root directory.
dirstat	Gets file status information for multiple files in a directory.
fchmod	Changes file access permissions.
fchown	Changes the owner and group IDs of files associated with a file descriptor.
fcntl	Controls open file descriptors and sockets.
fstat, fstatx	Provides information about a file associated with a file descriptor.
link	Creates an additional directory entry for an existing file.
mkdir	Creates a new directory.
mknod	Creates a special file that describes a device or a FIFO.
mknodx	Creates a special file that describes a device on a specific TCF cluster site.
mount	Mounts a file system.
readlink	Reads the contents of a symbolic link.
rename	Renames a directory or a file within a file system.
rmdir	Removes a directory.
rmslink	Removes a symbolic link.
stat, statx	Provides information about a file associated with a path name.
symlink	Creates a symbolic link to a file or directory.
sync	Forces all changes in the file system to be written to disk.
umask	Sets the file creation mask.
umount	Unmounts a file system.
unlink	Removes a directory entry.
ustat	Gets file system statistics.
utime	Set the access and modification times of a file.

1.2.2.3 Process Control

The following system calls control creating, operating, and stopping processes:

brk	Changes the data segment space allocation.
disclaim	Disclaims content of a memory address range.
exec	Replaces the current process image with a new program.
exit	Terminates the current process.
fork	Creates and starts a child process.
getpriority	Gets program scheduling priority.
migrate	Moves a process to another cluster site.
nice	Determines or changes the execution priority of a process.
plock	Locks a process in memory.
profil	Starts and stops execution profiling.
ptrace	Traces execution of a child process.
rexec	Replaces the current process image with a new program on another site.
rfork	Creates and starts a child process on another cluster site.
run	Creates and starts a child process that runs a new program on a cluster site.
sbrk	Changes data segment space allocation.
setpriority	Sets program scheduling priority.
wait	Waits for a child process to stop or terminate.
waitpid	Obtains status information pertaining to a child process.
wait3	Waits for a child process to stop or terminate.

1.2.2.4 Process Identification

The following system calls get and set the IDs and limits of a process:

getegid	Gets the effective group ID.
geteuid	Gets the effective user ID.
getgid	Gets the real group ID.
getgroups	Gets the group access list.
getlocal	Gets the name of the **<LOCAL>** file system of a process.
getpgrp	Gets the process group ID.
getpid	Gets the process ID.
getppid	Gets parent process ID.
getspath	Gets the process site execution path.
gettimeofday	Gets the current time in microsecond precision.
getuid	Gets the real user ID.
getxperm	Gets the process site execution permission.
setgid	Sets the real and effective group IDs.
setgroups	Sets the group access list.
setlocal	Manages the **<LOCAL>** alias.
setpgid	Sets the process group ID.
setpgrp	Sets the process group ID.
setsid	Sets the session ID.
setspath	Sets the process site execution path.
settimeofday	Sets the current time in microsecond precision.
setuid	Sets the real and effective user IDs.
setxperm	Sets the process site execution permission.
setxuid	Uses real **uid** or **gid** on subsequent invocations of **exec** or **run**.
site	Gets the execution site of a process.
stime	Sets the current time.
sysconf	Retrieves the value of a system limit or option.
tcgetpgrp	Gets the foreground process group ID.
tcsetpgrp	Sets the foreground process group ID.
time	Gets the current time.
times	Gets process and child process times.
ulimit	Gets and sets the process's user limits.
uname	Gets the name of the current AIX system.
unamex	Gets the name of the current AIX system in binary form.
usrinfo	Gets and sets user information about the owner of a process.

1.2.2.5 System Administration

The following system calls support overall system operation.

acct	Enables and disables process accounting.
reboot	Causes a system reboot.
swapctl	Controls swap devices.

1.2.2.6 Cluster Communication

The following system calls support communication between cluster sites:

getsites Gets information about other cluster sites.
netctrl Gets and sets cluster communication parameters.
probe Probes another cluster site.

1.2.2.7 File System Replication

The following system calls support replicated file systems:

chfstore Changes the replication attributes of a file.
chlwm Changes the low water mark of a replicated file system.
dustat Gets file system statistics.
raccept Waits until file system recovery is required.
spropin Causes the latest version of a file to be propagated.

1.2.2.8 TCP/IP Communication

The following system calls support TCP/IP communication:

accept Accepts an incoming connection.
bind Binds a name to a socket.
connect Connects to a socket.
gethostname Gets the internet name of the host.
getsockname Gets the name of a socket.
getsockopt Gets the options of a socket.
gethostid Gets the internet address of a host.
listen Listens for connections on a socket.
receive Receives data from a socket.
sethostname Sets the internet name of the host.
setsockopt Sets the options of a socket.
sethostid Sets the internet address of a host.
send Sends data to a socket.
shutdown Shuts down a socket connection.
socket Creates a socket.
socketpair Creates a pair of connected sockets.

1.2.2.9 Signals

Signals are sent to processes when exceptional events occur. A signal interrupts the activity that a process is performing and causes it to take a special action. For example, when a user presses the **Alt-Pause** key sequence at a workstation, the **SIGINT** signal is sent to the user's processes. Normally, this causes them to terminate, but each process can arrange to ignore the signal, or to take some other action. The signals that can occur are defined in the **sys/signal.h** header file, and they are further described in "sigaction, sigvec, signal" in topic 1.2.263.

Standard signal processing is compatible with UNIX System V and is described in more detail in "sigaction, sigvec, signal" in topic 1.2.263. The following system calls handle standard signal processing:

alarm	Sets the process's alarm clock.
kill	Sends a signal to one or more processes.
pause	Suspends the process until a signal arrives.
signal	Sets the action to take when the process receives a signal.

Enhanced signal processing adds several useful features to the facility. It is described in more detail in "sigaction, sigvec, signal" in topic 1.2.263. The following system calls control enhanced signal processing:

sigaction	Sets the action to take when the process receives a signal.
sigpending	Examines pending signals.
sigprocmask	Sets or changes the process signal mask.
sigreturn	Returns from a signal handler.
sigstack	Specifies an alternate stack upon which to process signals.
sigsuspend	Atomically changes the set of blocked signals and waits for an interrupt.

1.2.2.10 Semaphores, Message Queues, and Shared Memory Segments

In addition to signals, the AIX Operating System provides three facilities that provide flexible interprocess communication (IPC): semaphores, message queues, and shared memory segments. Details about the philosophy and use of each these facilities is beyond the scope of this book.

The following system calls deal with message queues, semaphores, or shared memory:

msgctl **semop**
msgget **shmat**
msgop **shmctl**
semctl **shmdt**
semget **shmget**

All three facilities are accessed in a similar manner. The steps are outlined here in approximately the order that they appear in programs:

1. The user specifies a **key** to identify the individual semaphore set or the message queue or shared segment to be accessed. This key is analogous to a file name in that it has been previously agreed upon to identify a specific data structure.

   The key **IPC_PRIVATE** (defined in the **sys/ipc.h** header file) is a special key value that specifies that the data structure is to be private to the current process.

   Keys can be generated by any algorithm as long as the same algorithm is used by all processes on the system. The **ftok** subroutine provides a standard algorithm for generating IPC keys. (See " stdipc: ftok" in topic 1.2.284 for information about this subroutine.)

2. System calls whose names end with -**get** (**semget**, **msgget**, and **shmget**) use the key to obtain access to the requested data structure. The -**get** system calls are analogous to **open**: each returns an integer identifier (analogous to a file descriptor) that identifies the data structure for access with other system calls.

   Normally, if the semaphore, message queue, or shared segment does not already exist, then the -**get** system call creates the necessary data structure. If another process has already created the data structure by calling the same -**get** system call with the same **key**, then the identifier of that data structure is returned. This action can be modified with the **semflg**, **msgflg**, or **shmflg** parameter.

   However, if **IPC_PRIVATE** is specified as the key, a private data structure is created. No key exists with which to identify this data structure, so only processes that have its identifier can access it. The current process must pass the identifier to other processes that are to access it. For example, the identifier can be passed to a child process through the **argv** argument vector (see "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71 for details).

3. Shared memory segments must next be attached using the **shmat** system call.

4. The **semop** system call accesses semaphores. Message queues are accessed by **msgsnd**, **msgrcv**, and **msgxrcv**. Programs can access shared memory segments as regular memory through the pointer returned by the

shmat system call.

5. System calls whose names end with -**ctl** (**semctl**, **msgctl**, and **shmctl**) perform a variety of control operations on the data structure. These control operations include getting status information and changing the access permissions. The data structure associated with each type of IPC identifier is defined in the description of the corresponding -**ctl** system call.

6. When no longer in use, shared memory segments must be detached using the **shmdt** system call.

7. The IPC identifier and the associated data structure should then be removed from the system with the **IPC_RMID** operation of the corresponding -**ctl** system call.

Each IPC data structure contains an **ipc_perm** structure, which contains access permission information. The **ipc_perm** structure is defined in the **sys/ipc.h** header file, and it contains the following members:

```
ushort uid;   /* Owner's user ID */
ushort gid;   /* Owner's group ID */
ushort cuid;  /* Creator's user ID */
ushort cgid;  /* Creator's group ID */
ushort mode;  /* Access permission mode */
ushort seq;   /* Slot usage sequence number */
key_t  key;   /* Key */
```

The access permission mechanism resembles the one for files, except that execute permission does not exist for IPC facilities. The **semget**, **msgget**, and **shmget** system calls set the initial permissions when they create new IPC data structures. Also, the user (group) permissions apply if the process's effective user (group) ID matches either **uid** (**gid**) or **cuid** (**cgid**). The permissions can be changed with the corresponding -**ctl** system calls. The **uid** and **gid** fields identify the user and group that own the file for determining whether a given process may access a data structure. The **cuid** and **cgid** fields identify the process that created the data structure, and they cannot be changed.

The **mode** field is constructed by logically ORing one or more of the following values. Note that these values are defined in the **sys/stat.h** header file and that they are a subset of the access permissions that apply to files.

S_IRUSR Permits the process that owns the data structure to read it.
S_IWUSR Permits the process that owns the data structure to modify it.
S_IRGRP Permits the group associated with the data structure to read it.
S_IWGRP Permits the group associated with the data structure to modify it.
S_IROTH Permits others to read the data structure.
S_IWOTH Permits others to modify the data structure.

For more information about the interprocess communication facilities, see *AIX Programming Tools and Interfaces*.

1.2.3 Subroutines

Each subroutine entry contains a "Library" section that indicates the library where the subroutine is stored. Subroutines are stored in libraries to conserve storage space and to make the program linkage process more efficient. A ***library*** (sometimes called an ***archive***) is a data file that contains copies of a number of individual files and control information that allows them to be accessed individually.

The libraries that contain the subroutines described in this book are located in the **/lib** and **/usr/lib** directories. By convention, all of them have names of the form **libname.a**, where **name** identifies the specific library.

You do not need to do anything special to use subroutines from the Standard C Library (**libc.a**) or the Run-time Services Library (**librts.a**). The **cc** command automatically searches these libraries for subroutines that a program needs. However, if you use subroutines from another library, you must tell the compiler to search that library. If your program uses subroutines from the library **libname.a**, compile your program with the flag **-lname**. The following example compiles the program **myprog.c**, which uses subroutines from the **libdbm.a**:

```
cc myprog.c -ldbm
```

You can specify more than one **-l** flag, but they must be specified **after** any other flags. See the **cc** command in *AIX Operating System Commands Reference* for details.

The libraries discussed in the book are:

Standard C Library **libc.a**)
Standard I/O Library **libc.a**)
Internet Library **libc.a**)
Run-time Services Library **librts.a**)
Math Library **libm.a**)
Programmers Workbench Library **libPW.a**)
Curses Library **libcurses.a**)
Extended Curses Library **libcur.a**)
Database Library **libdbm.a**)
Queued Backend Subroutine Library **libqb.a**)
Lex Library **libl.a**)
Yacc Library **liby.a**)
4.3BSD Compatibility Library **libbsd.a**)
Graphics Libraries **libplot.a**, **libprint.a**, **lib300.a**, **lib300s.a**, **lib4014.a**, **lib450.a**, **libdumb.a**, **librt0.a**)
Advanced Display Graphics Support Library **libgsl.a**)
Object File Access Routine Library **libld.a**)

The Standard I/O subroutines are actually contained in the Standard C Library (**libc.a**). These subroutines implement a buffered I/O system on top of the basic I/O provided by the system calls. For more information about these subroutines, see " stdio" in topic 1.2.283.

The Internet subroutines and system calls are also contained in the standard C library (**libc.a**). These subroutines and system calls support Internet protocols (IP). For more information on Internet subroutines and system calls, see "sockets library" in topic 1.2.277.

1.2.4 Syntax

The "Syntax" section of each system call and subroutine entry in this book gives the syntax needed to invoke it. The following conventions are used in this section:

Boldface type shows text to be entered exactly as shown.

Italic type shows parameters that should be replaced with actual values.

[ ] (square brackets) enclose optional parameters.

... (an ellipsis) follows a parameter that can be repeated any number of times.

The information shown in each "Syntax" section is usually the set of declarations as they might appear in the actual C language definition of the call or subroutine. These declarations give you more information than showing the exact calling sequence as it appears in a user program.

Consider the following example of a subroutine entry:

```
#include <stdio.h>

FILE *fopen (path, type)
char *path, *type;
```

The **#include** statement names a header file that contains definitions needed by the subroutine. See "Header Files" in topic 1.2.5 for more information.

The first line following the **#include** statement shows the data type of the return value (**FILE ***), the name of the subroutine (**fopen**), and the parameters that it takes (**path** and **type**). The following lines indicate the data type of each parameter. The fact that the name **FILE** is in all capitals indicates that this data type is defined in the **stdio.h** header file.

This subroutine might actually be used in a program like this:

```
#include <stdio.h>
...
main ( )
{
   FILE *inputfile;
   char filename [ ] = "test.data";
   ...
   inputfile = fopen (filename, "r+");
   ...
}
```

Note that the type of both parameters is stated as **char *** (pointer to character), but that the value given for each is actually a pointer to a character string (an array of characters). In the C language, pointers and arrays are treated similarly so that the notations ***p** and **p[0]** are generally interchangeable. Thus, when this book shows a parameter of type **char ***, a character string is frequently required. Check the "Description" section to make sure.

Because **fopen** returns a type other than **int**, the subroutine must be declared so that the compiler knows this information. In this particular case, it is already declared for you in the header file. Sometimes, however, you may need to declare a system call or subroutine yourself. For this example, the declaration would take the form:

```
FILE *fopen();
```

Such a declaration should be put before any references to the system call or subroutine. Note that the declaration resembles the syntax shown in this book, except that the parameters are omitted and a semicolon is added to the end.

See *AIX C Language User's Guide* and *AIX C Language Reference* or another C language manual for more detailed information about pointers, arrays, and subroutine declarations.

1.2.5 Header Files

Many system calls and subroutines require that header files be included in the programs that use them. When this is the case, the **#include** statements needed are shown in the "Syntax" section of the call or subroutine entry. Consider the following example:

```
#include <stdio.h>
```

When a program is being compiled, this **#include** statement inserts the text of the **stdio.h** header file into the source program. The **< >** delimiters indicate that the file is located in the **/usr/include** directory. All of the header files used by the system calls and subroutines described in this book are located in **/usr/include** or in one of its subdirectories.

The header files contain definitions of constants and macros that the C language preprocessor interprets. The **#include** statements must precede all references in your program to the constants and macros that the header files define. Most of the time you can simply put all of the **#include** statements together at the top of the program. If you use several calls or subroutines that require the same header files, then each file should be included only once. If a system call or subroutine requires more than one header file, be careful to enter the **#include** statements in the order shown.

By convention, the names of most of the constants are in capital letters. Therefore, a name that appears in this book in capitals (for example, EFAULT) is a constant defined in a header file. A few constants are not named in capitals, notably **stdin**, **stdout**, and **stderr**, which are defined in **stdio.h**.

The constant NULL is commonly used to denote a null pointer value. This book sometimes mentions NULL when discussing system calls and subroutines that do not require header files. If you compile a program and get an error message indicating that NULL is not defined, insert the following statement before the first NULL in your program:

```
#define  NULL  0
```

In addition to constants, header files sometimes define macros and data types. Macros take parameters and resemble subroutines, but there are several differences:

- You must not type a space between the macro name and the ope parenthesis that follows it. For example, the macro call **getc(...)** is valid, but **getc (...)** is not. The preprocessor does not recognize the second of these as a macro, and so it does not make the proper substitution.

- You cannot take the address of a macro with the C language **&** operator.

- The parameters of a macro are evaluated in a different manner fro those of a subroutine. See *AIX C Language User's Guide* and *AIX C Language Reference* or another C language manual for details about parameter evaluation.

Certain macros are implemented both as subroutines and macros. To use macros as subroutines, use the **#undef** C Preprocessor directive to undefine the macro (first include the appropriate header file).

1.2.6 a64l, l64a

Purpose
Converts between long integers and base-64 ASCII strings.

Library
Standard C Library (**libc.a**)

Syntax

```
                                        char *l64a (l)
        long a64l (s)                   long l;
        char *s;
```

Description
The **a64l** and **l64a** subroutines maintain numbers stored in base-64 ASCII characters. This is a notation in which long integers are represented by up to six characters, each character representing a digit in a base-64 notation.

The following characters are used to represent digits:

.	represents	0.
/	represents	1.
0--9	represent	2--11.
A--Z	represent	12--37.
a--z	represent	38--63.

Note: Base-64 ASCII strings' lowest bit is the left-most bit, not the right-most bit.

The **a64l** subroutine takes a pointer to a null-terminated character string containing a value in base-64 representation and returns the corresponding **long** value. If the string pointed to by the **s** parameter contains more than six characters, the **a64l** subroutine uses only the first six. The **a64l** subroutine does not check for invalid ASCII characters such as @ or %. Also, for any base-64 ASCII string larger than zzzzz/ or 2147483647, the **a64l** subroutine does not return the expected value.

Conversely, the **l64a** subroutine takes a **long** parameter and returns a pointer to the corresponding base-64 representation. If the **l** parameter is 0, then the **l64a** subroutine returns a pointer to a null string.

For any base-64 ASCII string larger than zzzzz/ or 2147483647, the **l64a** subroutine does not return the expected value. The value returned by **l64a** is a pointer into a static buffer, the contents of which are overwritten by each call.

1.2.7 abort

Purpose
Generates a **SIGABRT** signal to terminate the current process.

Library
Standard C Library (**libc.a**)

Syntax

int abort ( )

Description
The **abort** subroutine causes a **SIGABRT** signal to be sent to the current process, usually terminating the process and producing a memory dump.

It is possible for the **abort** subroutine to return control if **SIGABRT** is caught or ignored. In this case, **abort** returns the value returned by the **kill** system call.

If **SIGABRT** is neither caught nor ignored, and if the current directory is writable, the **abort** subroutine produces a memory dump in a file named **core** in the current directory. The shell then displays the message:

```
abort - core dumped
```

Related Information
In this book: "exit, _exit" in topic 1.2.73, "kill, kill3, killpg" in topic 1.2.138, and "sigaction, sigvec, signal" in topic 1.2.263.

The **dbx** command in *AIX Operating System Commands Reference*.

1.2.8 abs

Purpose
Returns the absolute value of an integer.

Library
Standard C Library (**libc.a**)

Syntax

```
int abs (i)
int i;
```

Description
The **abs** subroutine returns the absolute value of its integer operand.

Note: A twos-complement integer can hold a negative number whose absolute value is too large for the integer to hold. When given this largest negative value, the **abs** subroutine returns the same value.

Related Information
In this book: "floor, ceil, fmod, fabs, rint" in topic 1.2.81.

1.2.9 accept

Purpose
Accepts a connection on a socket.

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>

int accept (s, addr, addrlen)
int s;
struct sockaddr *addr;
int *addrlen;
```

Description
The **accept** system call extracts the first connection on the queue of pending connections, creates a new socket with the same properties as **s**, and allocates a new file descriptor for that socket. If no pending connections are present on the queue and the calling socket is not marked as nonblocking, **accept** blocks the caller until a connection is present. If the socket specified by **s** is marked nonblocking and there are no connections pending on the queue, **accept** returns an error as described below. The accepted socket cannot be used to accept more connections. The original socket, **s**, remains open and can accept more connections.

The **s** parameter is a socket that was created with the **socket** system call, was bound to an address with the **bind** system call, and has issued a successful call to the **listen** system call.

The **addr** parameter is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of **addr** is determined by the domain in which the communication occurs. The **addrlen** parameter initially contains the amount of space pointed to by the **addr** parameter. On return, it contains the actual length (in bytes) of the address returned. This system call is used with connection-based socket types, such as **SOCK_STREAM**.

Before calling the **accept** system call, you can find out if the socket is ready to accept the connection by doing a read select with the **select** system call.

Return Value

Upon successful completion, the nonnegative socket descriptor of the accepted socket is returned. A socket marked nonblocking with the **O_NONBLOCK** flag or the **FIONBIO ioctl** returns a value of -1 and sets **errno** to **EAGAIN** in the situation where is would otherwise have blocked.

A socket marked with the **O_NDELAY** flag returns ZERO in the situation where it would otherwise have blocked.

Error Conditions
The system call fails if one or more of the following are true:

EBADF The **s** parameter is not valid.

ENOTSOCK The **s** parameter refers to a file, not a socket.

EOPNOTSUPP The referenced socket is not of type **SOCK_STREAM**.

EFAULT The **addr** parameter is not in a writable part of the user address space.

EAGAIN The socket is marked nonblocking with the **O_NONBLOCK** flag or the **FIONBIO ioctl** in the situation where it would otherwise have blocked.

Related Information
In this book: "bind" in topic 1.2.20, "connect" in topic 1.2.49, "listen" in topic 1.2.157, "select" in topic 1.2.242, and "socket" in topic 1.2.275.

1.2.10 access

Purpose
Determines the accessibility of a file.

Syntax

```
#include <unistd.h>

int access (path, amode)
char *path;
int amode;
```

Description
The **access** system call checks the accessibility of the file specified by the **path** parameter. If **path** refers to a symbolic link, the **access** system call returns information about the file pointed to by the symbolic link.

Access permission to all components of the **path** parameter is determined by using the real user ID instead of the effective user ID and the concurrent group set (without the effective group ID) along with the real group ID.

The bit pattern contained in **amode** is constructed by logically ORing the following values:

R_OK	Checks read permission.
W_OK	Checks write permission.
X_OK	Checks execute (search) permission.
F_OK	Checks to see if the file exists.

The owner of a file has access checked with respect to the owner read, write, and execute mode bits. Members of the file's group other than the owner have access checked with respect to the group mode bits. All others have access checked with respect to the other mode bits.

Return Value
If the requested access is permitted, a value of 0 is returned. If the requested access is denied, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
Access to the file is denied if one or more of the following are true:

ENOTDIR A component of the path prefix is not a directory.

ENOENT Read, write, or execute (search) permission is requested for a null path name.

ENOENT The named file does not exist.

EACCES Search permission is denied on a component of the path prefix.

EACCES Permission bits of the file mode do not permit the requested access.

EROFS Write access is requested for a file on a read-only file system.

ETXTBSY Write access is requested for a pure procedure (shared text) file that is being executed.

EFAULT The **path** parameter points to a location outside of the process's allocated address space.

ESTALE The process's root or current directory is located in an NFS virtual file system that has been unmounted.

ENAMETOOLONG
A component of the **path** parameter exceeded **NAME_MAX** characters or the entire **path** parameter exceeded **PATH_MAX** characters.

ENOENT A hidden directory was named, but no component inside it matched the process's current site path list.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

ELOOP A loop of symbolic links was detected.

ENFILE The system inode table is full.

If the Transparent Computing Facility is installed on your system, **access** can also fail if one or more of the following are true:

ESITEDN1 **path** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE **path** is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of **path** is replicated but is not stored on any site which is currently up.

EROFS Write access is requested for a file on a replicated file system in which the primary copy is unavailable.

EINTR A signal was caught during the **access** system call. This can occur if the internal open of this file is suspended during topology change.

Related Information
In this book: "chmod, fchmod" in topic 1.2.44 and "statx, fstatx, stat, fstat, fullstat, ffullstat, lstat" in topic 1.2.282.

1.2.11 acct

Purpose
Enables and disables process accounting.

Syntax

int acct (path)
char *path;

Description
The **acct** system call enables the accounting routine when the **path** parameter specifies a valid path name of a file and no errors occur during the system call. The path name specifies the file to which accounting records are written for each process as it terminates. (For information about the accounting file, see "acct" in topic 2.3.3.) When the **path** parameter is 0 or NULL, the **acct** system call disables the accounting routine.

Warning: To ensure accurate accounting, each node must have its own accounting file, which can be located on any node in the network.

The effective user ID of the calling process must be superuser to use the **acct** system call.

Return Value
Upon successful completion, **acct** returns a value of 0. If **acct** fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **acct** system call fails if one or more of the following are true:

EPERM The effective user ID of the calling process is not superuser.

EBUSY An attempt is made to enable accounting when it is already enabled.

ENOTDIR A component of the **path** parameter is not a directory.

ENOENT Any component of the accounting file's path name does not exist.

EACCES Any component of the **path** parameter denies search permission.

EACCES The file named by the **path** parameter is not an ordinary file.

EISDIR The named file is a directory.

EROFS The named file resides on a read-only file system.

EFAULT The **path** parameter points to a location outside of the process's allocated address space.

ENAMETOOLONG
A component of the **path** parameter exceeded **NAME_MAX** characters or the entire **path** parameter exceeded **PATH_MAX** characters.

ENOENT A hidden directory was named, but no component inside it matched the process's current site path list.

ENOENT A symbolic link was named, but the file to which it refers does

not exist.

ELOOP A loop of symbolic links was detected.

ENOSPC The file system is out of inodes.

ENFILE The system inode table is full.

ETXTBSY Write access is requested for a pure procedure (shared text) file that is being executed.

If the Transparent Computing Facility is installed on your system, **acct** can also fail if one or more of the following are true:

ESITEDN1 **path** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

EINTR A signal was caught during the **acct** system call.

ENOSTORE **path** is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of **path** is replicated but is not stored on any site which is currently up.

EROFS Write access is requested for a file on a replicated file system in which the primary copy is unavailable.

Related Information
In this book: "acct" in topic 2.3.3, "exit, _exit" in topic 1.2.73 and "sigaction, sigvec, signal" in topic 1.2.263.

The **acct** command in *AIX Operating Commands Reference*.

1.2.12 acosh, asinh, atanh

Purpose
Inverse hyperbolic functions.

Library
Math Library (**libm.a**)

Syntax

```
#include <math.h>

double acosh (x)
double x;

double asinh (x)
double x;

double atanh (x)
double x;
```

Description
These subroutines acosh, asinh, atanh compute the designated inverse hyperbolic functions for real arguments.

The **acosh** subroutine returns the value HUGE and sets error to ED8M if the argument is less than 1.

The **atanh** subroutine returns the reserved operand if the argument has absolute value bigger than or equal to 1.

Related Information
In this book: "cbrt, exp, expm1, log, log10, log1p, pow, sqrt" in topic 1.2.28, and "math.h" in topic 2.4.13.

1.2.13 adjtime

Purpose
Corrects the time to allow synchronization of the system clock.

Syntax

```
#include <sys/time.h>

adjtime (delta, olddelta)
struct timeval *delta;
struct timeval *olddelta;
```

Description
The **adjtime** system call makes small adjustments to the system time, as returned by **gettimeofday**, advancing or retarding it by the **timeval delta**. If **delta** is negative, the clock is slowed down by incrementing it more slowly than normal until the correction is complete. If **delta** is positive, a larger increment than normal is used. The skew used to perform the correction is generally a fraction of one percent. Thus, the time is always a monotonically increasing function. A time correction from an earlier call to **adjtime** may not be finished when **adjtime** is called again. If **olddelta** is nonzero, the structure pointed to contains, upon return, the number of microseconds still to be corrected from the earlier call.

This call may be used by time servers that synchronize the clocks of computers in a local area network. Such time servers would slow down the clocks of some machines and speed up the clocks of others to bring them to the average network time.

The call **adjtime** is restricted to use by the superuser.

Return Value
Upon successful completion, a value of 0 is returned. If the **adjtime** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **adjtime** system call fails if one or more the following are true:

EFAULT The **delta** or **olddelta** parameters point to a location outside of the process's allocated address space.

EPERM The effective user ID of the calling process is not superuser.

Related Information
In this book: "gettimeofday, settimeofday, ftime" in topic 1.2.123.

The **date** command in *AIX Operating System Commands Reference*.

The discussion of **timed** and **timedc** in *AIX TCP/IP User's Guide*.

1.2.14 alarm

Purpose
Sets a process's alarm clock.

Syntax

unsigned int alarm (secs)
unsigned int secs;

Description
The **alarm** subroutine sets a timer which causes a **SIGALRM** signal to be delivered to the calling process after the number of real-time seconds specified by the **secs** parameter have elapsed. After the **SIGALRM** signal is delivered the timer is turned off.

Unless caught, blocked, or ignored the **SIGALRM** signal will cause the process to terminate. (See "sigaction, sigvec, signal" in topic 1.2.263 for more information about signals.)

Alarm requests are not stacked; successive calls reset the alarm timer. If the **secs** parameter is 0 the alarm timer is turned off, cancelling any pending alarm request.

Because of process scheduling delays, the actual time lapse between setting the timer and getting the **SIGALRM** signal may be more than requested.

The alarm timer value is inherited through the **exec** system call. The child process after a **fork** system call will have its alarm timer turned off.

If the Transparent Computing Facility is installed, the alarm timer is preserved through **migrate** and **rexec** system calls and during migrates caused by **SIGMIGRATE**.

The **alarm** subroutine is a simplified interface to the **setitimer** system call. See "getitimer, setitimer" in topic 1.2.101 for more information.

Return Value
The **alarm** subroutine returns the number of seconds previously remaining on the alarm timer, or 0 if the timer was not running. (Note that because timer values are inherited through **exec**, the timer may already be running when a program begins to execute.)

Related Information
In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, "getitimer, setitimer" in topic 1.2.101, "pause" in topic 1.2.202, and "sigaction, sigvec, signal" in topic 1.2.263 .

1.2.15 alphasort

Purpose
Provides a comparison function for sorting alphabetically.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <dirent.h>

int alphasort (dir1, dir2)
struct dirent **dir1, **dir2;
```

Description
The **alphasort** subroutine alphabetically compares the **d_name** members of the two **dirent** structures pointed to by the **dir1** and **dir2** parameters. This subroutine can be passed as the **compar** parameter to either the **scandir** or **qsort** subroutine, or a user-supplied subroutine can be used instead. (See "qsort" in topic 1.2.217 and "scandir" in topic 1.2.240 for more information.)

Return Value
The **alphasort** subroutine returns the following values:

Less than 0	If the **d_name** member pointed to by the **dir1** parameter is lexically less than the **d_name** member pointed to by the **dir2** parameter.
0	If the **d_name** members pointed to by **dir1** and **dir2** are equal.
Greater than 0	If the **d_name** member pointed to by the **dir1** parameter is lexically greater than the **d_name** member pointed to by the **dir2** parameter.

Related Information
In this book: "qsort" in topic 1.2.217 and "scandir" in topic 1.2.240.

1.2.16 assert

Purpose
Verifies a program assertion.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <assert.h>

void assert (expression)
int  expression;
```

Description
The **assert** macro puts diagnostics into a program. If **expression** is false (zero), then **assert** writes the following message on the standard error output and aborts the program:

Assertion failed: expression, file filename, line linenum

In the error message, **filename** is the name of the source file and **linenum** is the source line number of the **assert** statement.

If you compile a program with the preprocessor option **-DNDEBUG**, or with the preprocessor control statement **#define NDEBUG** ahead of the **#include <assert.h>** statement, assertions will not be compiled into the program.

Related Information
In this book: "abort" in topic 1.2.7.

The **cpp** command in *AIX Operating System Commands Reference*.

1.2.17 async_daemon

Purpose
Performs asynchronous block input/output operations for an NFS client.

Syntax
void async_daemon ()

Description

The **async_daemon** system call performs the asynchronous block I/O operations for NFS clients. If NFS is installed on your system, **async_daemon** is called by the **biod** command from the **/etc/rc.nfs** file when the system starts. Once started, **async_daemon** loops continuously and does not return.

Related Information
The **biod** command in *AIX Operating System Commands Reference*.

The NFS configuration section in *Managing the AIX Operating System*.

1.2.18 bcmp, bzero, ffs

Purpose
Performs byte operations on strings of variable length.

Library
Standard C Library (**libc.a**)

Syntax

```
int bcmp (string1, string2, intgffs (integer)
char *string1, *string2;    int integer;
int length;

void bzero (string, length)
char *string;
int length;
```

Description
These functions perform various operations on variable length strings of bytes. They do not check for null bytes as the routines in **string**. (See "string" in topic 1.2.288 for a description of those routines.)

The **bcmp** subroutine compares the bytes in **string1** with the bytes in **string2**. The **bcmp** subroutine returns a 0 value if the two strings are identical and a nonzero value otherwise. Both strings are assumed to be **length** bytes long.

The **bzero** subroutine places the number of bytes set to 0 that the **length** parameter specifies into the string pointed to by **string**.

The **ffs** subroutine finds the first set bit in the **integer** passed to it and returns the number of that bit. Bits are numbered starting at 1 from the least significant bit. A return value of 0 indicates that the value passed is 0.

For more information on the **memcmp** and **memset** subroutines, which can be used to perform the same functions as **bcmp** and **bzero**, respectively, and on the **bcopy** subroutine, see "memory: memccpy, memchr, memcmp, memcpy, memset, bcopy" in topic 1.2.166.

Related Information
In this book: "memory: memccpy, memchr, memcmp, memcpy, memset, bcopy" in topic 1.2.166 and "string" in topic 1.2.288.

1.2.19 bessel: j0, j1, jn, y0, y1, yn

Purpose
Computes Bessel functions.

Library
Math Library (**libm.a**)

Syntax

```
#include <math.h>

double j0 (x)                    double y0 (x)
double x;                        double x;

double j1 (x)                    double y1 (x)
double x;                        double x;

double jn (n, x)                 double yn (n, x)
int n;                           int n; double x;
double x;
```

Description
The **j0** and **j1** subroutines return Bessel functions of **x** of the first kind, of orders 0 and 1, respectively. **jn** returns the Bessel function of **x** of the first kind of order **n**.

The **y0** and **y1** subroutines return the Bessel functions of **x** of the second kind, of orders 0 and 1, respectively. **yn** returns the Bessel function of **x** of the second kind of order **n**. The value of **x** must be positive.

Non-positive parameters cause **y0**, **y1**, and **yn** to return the value HUGE, to set **errno** to EDOM, and to write a message to the standard error output indicating a DOMAIN error.

Parameters that are too large in magnitude cause **j0**, **j1**, **y0**, and **y1** to return as much of the result as possible, to set **errno** to ERANGE, and to write a message to the standard error output indicating a TLOSS error.

You can change these error-handling procedures with the **matherr** subroutine.

Error Conditions
The **j0**, **j1**, **jn**, **y0**, **y1**, and **yn** subroutines fail if one or more of the following is true:

EDOM The value of **x** is NaN.

ERANGE The value of **x** was too large in magnitude.

Related Information
In this book: "matherr" in topic 1.2.163.

1.2.20 bind

Purpose
Binds a name to a socket.

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>

int bind (s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

Description

The **bind** system call assigns a name to an unnamed socket. When a socket is created with the **socket** system call, it belongs to the address family specified in the **socket** call, but has no name assigned yet. The **bind** system call requests that **name** be assigned to the socket.

Note that all named sockets must have unique names. A socket does not have to have a name before it can make a connection to another socket, and a socket returned by the **accept** system call already has a name assigned to it by the call.

Note: Sockets in the **AF_UNIX** address family create a name in the file system name space that must be deleted by the caller (using **unlink**) when it is no longer needed.

Note: If the Transparent Computing Facility is installed, socket naming is not unique across the cluster; each machine has its own name space of sockets.

Return Value
Upon successful completion, a value of 0 is returned. If the **bind** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The system call fails if one or more of the following are true:

EBADF The **s** parameter is not valid.

ENOTSOCK The **s** parameter refers to a file, not a socket.

EADDRNOTAVAIL
The specified address is not available from the local machine.

EADDRINUSE
The specified address is already in use.

EINVAL The socket is already bound to an address.

EACCES The requested address is protected, and the current user does not have permission to access it.

EFAULT The **name** parameter points to a location outside of the process's allocated address space.

EINVAL The *namelen* parameter exceeds MLEN (see **<sys/mbuf.h>**).

The following errors are specific to binding names in the **AF_UNIX** address family.

ENOTDIR A component of the path name is not a directory.

ENOENT A prefix component of the path name does not exist.

ELOOP A loop of symbolic links was detected.

EIO An I/O error occurred while making the directory entry or allocating the inode.

EROFS The name would reside in a read-only file system.

EISDIR A null path name was specified.

ENOSPC The file system is out of inodes, or the directory in which the socket is to be added does not have room for the new entry and cannot be extended.

EACCES Search permission is denied for a component of the path.

EACCES The directory in which the file is to be created does not permit writing.

If the Transparent Computing Facility is installed on your system, **bind** can also fail if one or more of the following are true:

ESITEDN1 **path** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE *path* is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of *path* is replicated but is not stored on any site which is currently up.

EROFS Write access is requested for a file on a replicated file system in which the primary copy is unavailable.

EINTR A signal was caught during the system call.

Related Information
In this book: "connect" in topic 1.2.49, "getsockname" in topic 1.2.120, "listen" in topic 1.2.157, and "socket" in topic 1.2.275.

1.2.21 brk, sbrk

Purpose
Changes data segment space allocation.

Syntax

```
                                        char *sbrk (incr)
int brk (endds)                         int incr;
char *endds;
```

Description
The **brk** and **sbrk** system calls dynamically change the amount of space allocated for the calling process's data segment. (For information about data segments, see "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71.)

The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the current end of the data segment. The amount of allocated space increases as the break value increases. The newly allocated space is initialized to 0. The break value can be automatically rounded up to a size appropriate for the memory management architecture.

The **brk** system call sets the break value to the value of the **endds** parameter and changes the allocated space accordingly.

The **sbrk** system call adds to the break value the number of bytes contained in the **incr** parameter and changes the allocated space accordingly. The **incr** parameter can be a negative number, in which case the amount of allocated space is decreased.

Return Value
Upon successful completion, the **brk** system call returns a value of 0, and the **sbrk** system call returns the old break value. If the **brk** or the **sbrk** system calls fail, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **brk** and the **sbrk** system calls fail and the allocated space remains unchanged if one or more of the following are true:

ENOMEM The requested change will allocate more space than is allowed by a system-imposed maximum. (For information on the system-imposed maximum on memory space, see "getrlimit, setrlimit, vlimit" in topic 1.2.115 and "ulimit" in topic 1.2.313.)

ENOMEM The requested change will set the break value to a value greater than the lowest stack address or to a value greater than the start address of any attached shared memory segment. (For information on shared memory operations, see "shmat" in topic 1.2.258, "shmdt" in topic 1.2.260, and "shmget" in topic 1.2.261.)

EINVAL The **endss** or **incr** parameters would free more data space than had been allocated.

Related Information
In this book: "exec: execl, execv, execle, execve, execlp, execvp" in

topic 1.2.71, "shmat" in topic 1.2.258, "shmdt" in topic 1.2.260, "getrlimit, setrlimit, vlimit" in topic 1.2.115, and "ulimit" in topic 1.2.313.

1.2.22 BSD4.3 library

Purpose
Describes the 4.3BSD functions provided by AIX.

Library
Standard C Library (**libc.a**)
Math Library (**libm.a**)
BSD Compatibility (**libbsd.a**)

Description
This section provides a list of 4.3BSD routines supported in AIX and information to help application programmers port 4.3BSD programs to AIX. Some of these routines can be found in **libbsd.a** and most of the rest in **libc.a**. The location and full description of each of the 4.3BSD routines can be found in the appropriate alphabetical location in this chapter.

Even if you are familiar with 4.3BSD programming, you should read this entire section before porting existing programs or writing new AIX programs that use BSD4.3 library functions. Once you have read this section, you can refer back to specific discussions or to the individual routine descriptions as needed.

Subtopics

1.2.22.1 BSD4.3 library Routines

The following table lists the 4.3BSD library routines that

1.2.22.2 Porting 4.3BSD Applications to AIX

This section provides programmers with the information necessary to use the 4.3BSD functions provided by AIX. It explains how to use the **libc.a** library and suggests some of the changes that may be needed to port 4.3BSD programs to AIX.

In general, when porting you should try to use makefile changes whenever possible, changing the original source code only when absolutely necessary. Changes you may want to make in the makefile include:

When compiling

Add a **-D_BSD** to your compile command to obtain 4.3BSD behavior in include files.

When linking

To access the BSD4.3 library routines, link with **/usr/lib/libbsd.a**.

An alternative to defining **-D_BSD** and **-lbsd** explicitly on the **cc** command line is to define the environment variable BSD in your shell. This environment variable instructs **cc** to place these items on the command line for you automatically.

If you must modify your source code, for the **cpp** C language preprocessor, enclose your changes in **ifdef** statements similar to the following:

```
#ifdef _BSD
        <code for 4.3BSD specific version>
#else
        <code for AIX specific version>
#endif
```

Note: The define for the AIX Operating System is **_AIX**.

There are two versions of **signal** and two versions of **sigvec**, one each in **libbsd.a** and **libc.a**. The versions of **signal** and **sigvec** in **libbsd.a** conform to 4.3BSD behavior: for example, caught signals are not reset, **signal** restarts certain system calls and **sigvec** restarts certain system calls by default. The versions of **signal** and **sigvec** in **libc.a** conform to the behavior of earlier versions of AIX.

The AIX signal implementation has been enhanced to support 63 signals through the **sigaction** system call. All 4.3BSD signals, except **SIGVTALRM**, have signal numbers less than or equal to 32 and can be used with **sigvec** and **signal**. **SIGVTALRM** has a signal number of 34 and must be used with the **sigaction** system call. See "sigaction, sigvec, signal" in topic 1.2.263 for more information.

Subtopics
1.2.22.2.1 4.3BSD Include Files
1.2.22.2.2 Specific Information on BSD4.3 library Routines
1.2.22.2.3 4.3BSD TTY Devices

1.2.22.2.1 4.3BSD Include Files

For most applications, the include files on the following list contain the necessary 4.3BSD definitions and structures. Just be sure to use **-D_BSD** when compiling your programs.

Although the include file **sys/errno.h** contains definitions for ENOTEMPTY and EWOULDBLOCK, no AIX system call returns these error numbers. Instead, EAGAIN is returned where a 4.3BSD application would expect EWOULDBLOCK, and EEXIST is returned by the **rename** and **rmdir** system calls rather than ENOTEMPTY. If **_BSD** is defined prior to including **sys/errno.h**, EWOULDBLOCK is automatically defined to EAGAIN. Then place the following definitions in your program after including **sys/errno.h**:

```
#undef    ENOTEMPTY
#define   ENOTEMPTY    EEXIST
```

General Include Files

fcntl.h
math.h
sgtty.h
strings.h
sysexits.h
sys/dir.h
sys/file.h
sys/ioctl.h
sys/msgbuf.h
sys/param.h
sys/resource.h
sys/signal.h
sys/socket.h
sys/syslog.h
sys/time.h
sys/ttychars.h
sys/ttydev.h
sys/types.h
sys/uio.h
sys/wait.h

Network Include Files

arpa/ftp.h
arpa/inet.h
arpa/nameser.h
arpa/telnet.h
arpa/tftp.h
net/af.h
net/if.h
net/if_arp.h
netinet/in.h
netinet/in_systm.h
netinet/in_var.h
netdb.h
resolv.h
sys/un.h

1.2.22.2.2 Specific Information on BSD4.3 library Routines

A few of the 4.3BSD library routines function differently in AIX. This section provides you with details of these differences for a few routines, then gives you a list of individual routine descriptions that also contain information on such differences.

strcpyn Calls to this subroutine must be replaced with the **strncpy** subroutine.

strcatn Calls to this subroutine must be replaced with the **strncat** subroutine.

If your 4.3BSD program uses any of the subroutines on the following list, refer to the full description of that routine in this chapter before porting.

flock
killpg
setbuffer, setlinebuf
syslog
utimes

1.2.22.2.3 4.3BSD TTY Devices

The AIX TTY driver also supports the 4.3BSD TTY interfaces. See "termio" in topic 2.5.28 for more information.

If your 4.3BSD program uses the **curses** library, **libcurses.a**, TTY issues should be handled at that level. Note, however, that AIX uses the **terminfo** subroutine instead of the **termcap** subroutine that is used in 4.3BSD.

When writing or updating code that gets and uses **ptys**, keep in mind that:

AIX uses an extended naming convention for **ptys**.

In 4.3BSD, the master pseudo-terminal name is taken from the set **/dev/ptyp-r 0-9a-f**, and the slave pseudo-terminal name is taken from the set **/dev/ttyp-r 0-9a-f**. In AIX, the master pseudo-terminal is referred to as the **controller**; its name is taken from the set **/dev/ptyp-zA-Z 0-9a-f**. The slave pseudo-terminal is referred to as the **server**; its name is taken from the set **/dev/ttyp-zA-Z 0-9a-f**.

ptys may be obtained differently in AIX from 4.3BSD.

When **ptys** are configured into the system using the **devices** command, the user has the choice of enabling a **logger** (that is, a getty process) on the **server** side of a **pty**. This is not necessary (and must not be done) for remote login applications such as Telnet. However, other applications may expect a **logger** to be enabled. The **/etc/ports** file contains a stanza for each **pty** for which a **logger** has been enabled.

See the special file, "pty" in topic 2.5.21 for more information.

Related Information
In this book: "sockets library" in topic 1.2.277.

The descriptions of individual BSD commands in *AIX Operating System Commands Reference*.

The descriptions of individual BSD commands in *AIX TCP/IP User's Guide*.

1.2.23 bsearch

Purpose
Performs a binary search.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>

void *bsearch(key, base, nmemb, size, compar)
void *key, *base;
int nmemb;
size_t size;
int (*compar) (void*, void*);
```

Description
The **bsearch** subroutine is a binary search routine generalized from Donald E. Knuth's *The Art of Computer Programming*, Volume 3, 6.2.1, Algorithm B. (*) It returns a pointer into a table indicating where a datum is found.

The table must already be sorted in increasing order according to the provided comparison function **compar**. The **key** parameter points to the datum to be sought in the table. The **base** parameter points to the element at the base of the table. The **nmemb** parameter is the number of elements in the table. The **compar** parameter is a pointer to the comparison function, which is called with two parameters that point to the elements being compared.

The comparison function must compare its parameters and return a value as follows:

- If the first parameter is less than the second parameter, **compar** must return a value less than 0.
- If the first parameter is equal to the second parameter, **compar** must return 0.
- If the first parameter is greater than the second parameter, **compar** must return a value greater than 0.

The comparison function need not compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

The pointers **key** and **base** should be of type pointer-to-element, and cast to type pointer-to-character. Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

Return Value
If the key is found in the table, the **bsearch** returns a pointer to the element found. If the key cannot be found in the table, then **bsearch** returns the value NULL.

Related Information
In this book: "hsearch, hcreate, hdestroy" in topic 1.2.130, "lsearch, lfind" in topic 1.2.160, "qsort" in topic 1.2.217, and "tsearch, tdelete, twalk" in topic 1.2.309.

(*) Reading, Massachusetts: Addison-Wesley, 1981.

1.2.24 catclose

Purpose
Close a message catalog previously opened for access.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <nl_types.h>

int catclose (catd)
nl_catd catd;
```

Description
The **catclose** subroutine closes the message catalog associated with the catalog descriptor **catd**. Any memory associated with the catalog descriptor is freed up at this time.

Return Value
A 0 (zero) is returned upon successful close of the catalog; a -1 if an error occurred. A failure may be due to an invalid catalog descriptor.

Related Information
The **gencat** and **mkcatdefs** commands in *AIX Operating System Commands Reference*.

The message catalog description in *Managing the AIX Operating System*.

In this book: "catgets" in topic 1.2.25, "catgetmsg" in topic 1.2.26 and "catopen" in topic 1.2.27.

1.2.25 catgets

Purpose
Retrieve a message from an open message catalog.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <nl_types.h>

char *catgets (catd, set_num, msg_num, s)
nl_catd catd;
int set_num, msg_num;
char *s;
```

Description
The **catgets** subroutine retrieves a message from an open message catalog. The **catd** parameter is a catalog descriptor returned from a successful **catopen**, **set_num** and **msg_num** specify the set and message from the catalog to retrieve, and **s** is a default string to return if the specified message cannot be retrieved. A pointer to the message from the catalog is returned upon success. If the call fails for any reason, a pointer to the default string is returned.

Return Value
The message retrieved by the **catgets** subroutine is held in a static data location so the data is overwritten on successive calls to **catgets**. It is up to the user program to copy the message returned to a local buffer to save the message.

Related Information
The **gencat** and **mkcatdefs** commands in *AIX Operating System Commands Reference*.

The message catalog description in *Managing the AIX Operating System*.

1.2.26 catgetmsg

Purpose
Retrieve a message from an open message catalog into a buffer.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <nl_types.h>

char *catgetmsg (catd, set_num, msg_num, buf, buflen)
nl_catd catd;
int set_num, msg_num;
char *buf;
int buflen;
```

Description
The **catgetmsg** subroutine reads a message from an open message catalog into a buffer. The **catd** catalog descriptor is returned from a successful **catopen**, **set_num** and **msg_num** specify the set and message from the catalog to retrieve, and **buf** is a user defined buffer to store the message and **buflen** in bytes. The **catgetmsg** subroutine attempts to read up to **buflen-1** bytes of a message string into **buf**. The message is specified by the **set_num** and **msg_num** parameters. The message in **buf** is terminated with a NULL byte. **catgetmsg** does not split a multibyte character. This may lead to the message being truncated with up to **buflen-4** bytes placed in **buf** before the NULL byte.

Return Value
A pointer to **buf** is returned upon success of **catgetmsg**. A NULL pointer is returned upon any failure.

Related Information
The **gencat** and **mkcatdefs** commands in the *AIX Operating System Commands Reference*. The message catalog description in *Managing the AIX Operating System*.

1.2.27 catopen

Purpose
Open a message catalog.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <nl_types.h>

nl_catd catopen (filename, parm)
char *filename;
int parm
```

Description
The **catopen** subroutine opens a message catalog, which must be done before a message can be extracted using **catgets** or **catgetmsg**.

The catalog descriptor **nl_catd** is defined in **nl_types.h.**. The **filename** parameter is the name of the catalog to open and the **parm** parameter is reserved for future use and should be set to zero. The results of setting this field to any other value is undefined. If no / (slash) is in the filename, then **NLSPATH** is used to resolve the path. If there is one or more / (slash) in the filename, an absolute path is assumed.

Return Value
A valid catalog descriptor (**nl_catd**) is returned if the catalog is successfully opened; a -1 if an error occurred. The **catopen** routine may fail due to an invalid pathname for the catalog, or if more than **NL_MAXOPEN** catalogs are currently open.

Error Conditions
The **catopen** subroutine will fail if the following is true:

ENOMEM Insufficient storage space is available.

Related Information
In this book: "environment" in topic 2.4.6.

The **gencat** and **mkcatdefs** commands in *AIX Operating System Commands Reference*.

The message catalog description in *Managing the AIX Operating System*.

In this book: "catclose" in topic 1.2.24, "catgets" in topic 1.2.25 and "catgetmsg" in topic 1.2.26.

1.2.28 cbrt, exp, expm1, log, log10, log1p, pow, sqrt

Purpose
Computes exponential, logarithm, power, square root and cube root functions.

Library
Math Library (**libm.a**)

Syntax

#include <math.h>

double cbrt (x) **double x;**	**double log10 (x)** **double x;**
double exp (x) **double x;**	**double log1p (x)** **double x;**
double expm1 (x) **double x;**	**double pow (x, y)** **double x, y;**
double log (x) **double x;**	**double sqrt (x)** **double x;**

Description
The **cbrt** subroutine returns the cube root of **x**.

The **exp** subroutine to returns **e**(x).

The **expm1** subroutine returns log (1+**x**).

The **log** subroutine returns the natural logarithm of **x**,
ln x
. The value of **x** must be positive.

The **log10** subroutine returns the logarithm base 10 of **x**,
log sub 10 % x
. The value of **x** must be positive.

The **log1p** subroutine returns log (1+**x**).

The **pow** subroutine returns **x**(y). The values of **x** and **y** may not both be 0. If **x** is negative or 0, then **y** must be an integer.

The **sqrt** subroutine returns the square root of **x**. The value of **x** cannot be negative.

Note: The **expm1** and **log1p** subroutines are useful to guarantee that financial calculations of ((1+**x(n**)-1/ x (**expm1(n*** **log1p** (**x**))/**x** are accurate when **x** is very small (for example, small daily interest rates). They also make it easier to write accurate inverse hyperbolic functions.

Error Conditions
The **exp**, **log**, **log10**, and **sqrt** subroutines can perform either of the following types of error handling. The **pow** subroutine always handles

errors according to the second method. Both types of error handling allow you to define special actions to be taken when an error occurs.

1. For **cbrt**, **exp**, **log**, **log10**, and **sqrt**, exception handling will be performed according to ANSI/IEEE standard 754 for binary floating-point arithmetic by default.

   If a hardware floating-point processor is installed in your system, then using this option can provide greater performance in addition to IEEE exception handling. This mode instructs the C compiler to generate code that avoids the overhead of the math library subroutines by generating math coprocessor code in-line.

2. The second method is **matherr** error handling, as described on page 1.2.163. The default error-handling procedures for these subroutines are as follows:

   cbrt If **x** is negative, **cbrt** returns the value 0, sets **errno** to EDOM and writes an error message to the standard error output.

   exp If the correct value overflows, **exp** returns HUGE and sets **errno** to ERANGE.

   expm1 If the correct value overflows, **expm1** returns HUGE but does not modify **errno**.

   log If **x** is negative or 0, **log** returns the value -HUGE, sets **errno** to EDOM if **x** is negative and to ERANGE if **x** is 0, and writes an error message to the standard error output.

   log10 If **x** is negative or 0, **log10** returns the value -HUGE, sets **errno** to EDOM if **x** is negative and to ERANGE if **x** is 0, and writes an error message to the standard error output.

   log1p If **x** is less than -1, then **log1p** returns the value QNaN. If **x** equals -1, then **log1p** returns the value -HUGE. In neither case is **errno** set.

   pow If **x** is negative or 0 and **y** is not an integer, or if **x** and **y** are both 0, **pow** returns the value 0, sets **errno** to EDOM, and writes an error message to the standard error output. If the correct value overflows, **pow** returns HUGE and sets **errno** to ERANGE.

   sqrt If **x** is negative, **sqrt** returns the value 0, sets **errno** to EDOM, and writes an error message to the standard error output.

This second method of error handling is invoked by including the **-z** option on the **cc** command line.

Related Information
In this book: "hypot, cabs" in topic 1.2.132, "matherr" in topic 1.2.163, and "sinh, cosh, tanh" in topic 1.2.271.

1.2.29 cd

Purpose
Provides access to CD ROM files.

Syntax

```
int cdopen (path)
char *path;

int cdalias (alias, devname)
char *alias, *devname;

int cdread (cdfd, buf, nbytes)
int cdfd;
char *buf;
unsigned nbytes;

int cdclose (cdfd)
int cdfd;
```

```
#include <sys/types.h>

off_t cdlseek (cdfd, offset, w
int cdfd;
off_t offset;
int whence;


#include <sys/stat.h>

int cdstat (path, buf)
char *path;
struct stat *buf;
```

Description
The **cd** routines provide basic access to a compact disk read-only memory (CD ROM) device, at the same level as system calls. The routines are distinct from standard system calls because a CD ROM is not organized as a normal file system. The routines may be linked with a C program with the library option **-lcd**.

The **cdopen** routine opens the CD ROM file named by its path argument and returns an integer descriptor to be used in subsequent **cdread** calls. Note that letters in the file names on a CD ROM are restricted to upper case. However, **cdopen** translates its argument to upper case before attempting the open, so the path argument is effectively case-insensitive. If the path to **cdopen** is absolute, the first component must be an alias for a CD ROM device. If it is relative, the default device **/cd0** is assumed.

The **cdalias** routine is used to establish mount points for CD ROM devices. The alias argument is a name which may be used as the first element in paths in subsequent **cdopen** and **cdstat** calls. The **dev** argument is a device name to which the alias is mapped. After the call **cdalias ("/cd2","/dev/cd1");**, for example, a reference to **/cd2/x** as a CD ROM file name would refer to file **x** on the CD ROM device, **/dev/cd1**. There is one default alias, which maps **cd0** to **/dev/cd0**.

The **cdread** routine operates just as **read**. It attempts to read **nbytes** from the CD ROM file associated with **cdfd** into the buffer referenced by **buf**. The read starts at a position in the file give by a file pointer associated with **cdfd**. Upon return from **cdread**, this pointer is updated by the number of bytes actually read.

The **cdclose** routine closes the CD ROM file associated with **cdfd**.

The **cdlseek** routine alters the file position pointer associated with **cdfd** by the amount offset as follows:

If **whence** is 0, the pointer is set to offset bytes from the start of the file.

If **whence** is 1, the pointer is set to the current position plus offset, which may be negative to move backward.

If **whence** is 2, the pointer is set to the size of the file plus offset. The offset should not be greater than zero.

The **cdstat** routine is not implemented in the first phase. The **cdstat** routine may be used to obtain information about a CD ROM file. Information about the file named by **path** is returned in the structure referenced by **buf**. The **cdstat** routine performs the same internal mapping of letters to upper case as does **cdopen**. The structure used by **cdstat** is identical to that used by the normal **stat** request. However, some fields do not contain useful information due to the read-only nature of the device. A CD ROM file never shows write permission for any class of user. All three time stamps (creation, last modification and last access) are the same. Note that the user and group ID's of a file pertain to the environment in which it was recorded.

Return Value

If **cdopen**, **cdalias**, **cdlseek**, or **cdstat** fail, a value of -1 is returned and **errno** is set to indicate the error.

The **cdread** routine normally returns the number of bytes read. It returns 0 at EOF. On error, it returns -1 and sets **errno** to indicate the error.

Error Conditions
The **cdopen** routine fails if one or more of the following are true:

ENXIO Device associated with this special file *path* does not exist or the resulting device number is out of range.

EBUSY Device is busy.

EMFILE Maximum number of cdrom devices which are currently open.

ENOENT Failed to find the default alias name in the cdrom mount table.

The **cdalias** routine fails if one or more of the following are true:

EINVAL Alias name *alias* already in the alias list or alias name too long or incorrectly formed.

EINVAL Device name *devname* too long.

ENXIO Device name *devname* is not a character special file.

The **cdread** routine fails if one or more of the following are true:

EBADF The resulting file descriptor is not a valid open file descriptor.

EIO Physical I/O error occurred.

EINVAL A *nbytes* value of less than 0 is specified.

EFAULT The *buf* value points to a location outside of the process allocated address space.

The **cdclose** routine fails if the following are true:

EBADF The resulting file descriptor is not a valid open file

descriptor.

The **cdlseek** routine fails if one or more of the following are true:

EINVAL The *whence* value is invalid.

EINVAL The *offset* value is invalid.

EBADF The resulting file descriptor is not a valid open file descriptor.

The **cdstat** routine fails if one or more of the following are true:

ENXIO Device associated with this special file *path* does not exist or the resulting device number is out of range.

EBUSY Devise is busy.

EMFILE Maximum number of cdrom devices which are currently open.

ENOENT Failed to find the default alias name in the cdrom mount table.

File

/dev/cd0 Default CD ROM device.

Related Information
In this book: "cddir" in topic 1.2.30, "read, readv, readx" in topic 1.2.224, and "stat.h" in topic 2.4.22.

1.2.30 cddir

Purpose
Provides access to CD ROM directories.

Syntax

```
#include <sys/types.h>
#include <dirent.h>
#include <cddir.h>

cddir *cdopendir (path)
char *path;

struct dirent *cdreaddir (dirp)
cddir *dirp;

struct cdextdir *cdreadext (dirp)
cddir *dirp;

void cdrewinddir (dirp)
cddir *dirp;

void cdclosedir (dirp)
cddir *dirp;
```

Description
The **cddir** subroutines provide access to the directory structures on a compact disk, read-only memory (CD ROM) in the same fashion as the **directory** routines do for normal files. The CD ROM routines are distinct because a CD ROM is not organized as a normal file system. The routines may be linked with a C program with the library option **-lcd**.

The **cdopendir** routine opens the CD ROM directory named by **path** and returns a pointer to identify the directory for subsequent operations.

Directory entries may be read with either **cdreadext** or **cdreaddir**, depending on the type of information required. Successive calls read successive directory entries. **cdreadext** returns a pointer to a structure containing all information associated with a CD ROM file directory entry, while **cdreaddir** returns a pointer to a standard **dirent** structure. **cdreaddir** is provided for ease in converting code which currently uses the **directory** interface. Calls to **cdreadext** and **cdreaddir** may be freely intermixed.

Warning: These routines return pointers to static data areas which are overwritten by subsequent calls.

The **cdextdir** structure will change between phase 1 and phase 2 of development, since it has not yet been used with extended attribute CD ROM records.

cdrewinddir resets the position in the indicated directory to the beginning. A subsequent **cdreadext** or **cdreaddir** returns information about the first directory entry.

cdclosedir closes the indicated directory entry and releases associated internal data structures.

Files

/dev/cd0 default CD ROM device

/usr/lib/libcd.a CD ROM library

Related Information
In this book: "cd" in topic 1.2.29 and "directory: opendir, readdir, telldir, seekdir, rewinddir, closedir" in topic 1.2.60.

1.2.31 cfgadev

Purpose
Adds a device.

Library
Run-time Services Library (**librts.a**)

Syntax

```
#include <cfg01.h>

int cfgadev (master, system, xstanza, vstanza, dstanza, vflag, cflag)
char *master, *system, *xstanza, *vstanza, *dstanza;
int vflag, cflag;
```

Description
The **cfgadev** subroutine adds information about devices and device drivers to the system configuration.

The **master** parameter points to the full path name of the **master** file. The **system** parameter points to the full path name of the **system** file. These files are usually **/etc/master** and **/etc/system**, respectively.

The **xstanza**, **vstanza**, and **dstanza** parameters point to buffers that contain the text of attribute file stanzas. Any one or two of these parameters can be NULL pointers, indicating that a stanza of that type is not to be added, but at least one of them must point to a stanza buffer.

The **xstanza** parameter points to an AIX device driver stanza to be added to the **master** file. If the major device number is missing from the stanza, then the **cfgadev** subroutine generates a new one.

The **vstanza** parameter is not used but is retained for compatibility with the RT/AIX.

The **dstanza** parameter points to a device stanza to be added to the **system** file. It also generates a minor device number if only the prefix (c or b) is supplied or if the value is not unique.

The **vflag** parameter is either 1 (for yes) or 0 (for no), indicating whether to execute the **osconfig** command after the device stanza is added. If the **vflag** parameter is 1, then **cfgadev** executes the **osconfig** command with the **-a stname** flag, where **stname** is the name of the device stanza. The **osconfig** command then processes this stanza for driver addition and produces a shell procedure. The **cfgadev** subroutine then runs this shell procedure, which creates the special file **/dev/stname**, where **stname** is the name of the device stanza in the **system** file. If the **osconfig** command returns an error, then all stanzas that were added to the **master** and **system** files are deleted.

The **cflag** parameter is either 1 (for no) or 0 (for yes), for **osconfig** to call customize helpers when adding devices.

If the device stanza pointed to by the **dstanza** parameter contains the **specproc** keyword, then the program specified by the value of this keyword is executed to perform any special processing required when adding this device. The value of the **specproc** keyword must be the full path name of an executable file. The following arguments are passed to the program using the **argv** mechanism described in "exec: execl, execv, execle,

execve, execlp, execvp" in topic 1.2.71. All of them are passed as character strings.

argv[0] The full path name of the special-processing program
argv[1] The full path name of the **master** file
argv[2] The full path name of the **system** file
argv[3] The name of the device stanza
argv[4] The character string **"a"**, indicating addition.

If the special processing program fails, the device is still added to the system, but additional steps may be required before it can be used.

Return Value
Upon successful completion, the value **CFG_SFUL** is returned. If the **cfgadev** subroutine fails, then one of the following values is returned:

CFG_BFIC An input stanza is incomplete, or necessary information is missing.

CFG_BFNA A failure occurred while adding a stanza to the **master** or **system** file.

CFG_BFSM An input stanza buffer cannot be updated because the buffer is too small.

CFG_CLSE An error was detected while trying to close a file.

CFG_FCOR The **master** or **system** file is set up incorrectly.

CFG_MALF Memory allocation failed because of insufficient space.

CFG_MAXM The maximum number of minor device numbers has been reached for the driver associated with the device being added.

CFG_MPRE The prefix of the device's minor number is neither **b** nor **c**.

CFG_OPNE An error was detected while trying to open a file.

CFG_SLPF Special processing failed. The device was added but may require some additional steps before it can be used.

CFGT_VLNG A major device number could not be generated to complete an input stanza.

CFG_VCFG The **osconfig** command failed.

CFG_REBOOT AIX must be rebooted by calling the **cfgaply** routine, although **cfgadev** was successful. **cfgadev** will return **CFG_REBOOT** after a stanza is added to **/etc/master** with the mandatory attribute set to TRUE or after adding the first stanza associated with a device driver into **/etc/system**.

CFG_RBSLPF Special processing failed and a AIX reboot must be performed.

Related Information
In this book: "attributes" in topic 2.3.5, "master" in topic 2.3.32, and "system" in topic 2.3.56.

The **osconfig** command in *AIX Operating System Commands Reference*.

1.2.32 cfgaply

Purpose
Applies configuration information.

Library
Run-time Services Library (**librts.a**)

Syntax

```
#include <cfg03.h>

int cfgaply (restart)
int restart;
```

Description
The **cfgaply** subroutine rebuilds the AIX kernel by executing the **newkernel** command.

The **restart** parameter indicates whether to restart the system after the subroutine completes. If **restart** is a nonzero value, the system is restarted after completion.

Return Value
If the **restart** parameter is nonzero, the system is restarted, and the **cfgaply** subroutine does not return. If **restart** is 0 and **cfgaply** completes successfully, it returns the value **CFG_SUCC**. If the **cfgaply** subroutine itself fails, then the following value is returned:

CFG_AMKF The **newkernel** command failed.

Files

<LOCAL>/unix
<LOCAL>/unix.std
<LOCAL>/unix.last

Related Information

The **config** command in *AIX Operating System Commands Reference*.

The **newkernel** command in *Managing the AIX Operating System*.

1.2.33 cfgcadsz

Purpose
Adds or replaces a stanza in an attribute file.

Library
Run-time Services Library (**librts.a**)

Syntax

```
#include <cfg04.h>

int cfgcadsz (atfile, stanza, stname, after)
CFG__SFT *atfile;
char *stanza;
char *stname;
char *after;
```

Description
The **cfgcadsz** subroutine adds a new stanza or replaces an existing stanza in an attribute file. (For details about attribute files, see "attributes" in topic 2.3.5.)

The **atfile** parameter points to an open attribute file structure. The **stanza** parameter points to the buffer that contains the stanza to be written. The **stname** parameter points to the name of the stanza to be added to the file.

The **after** parameter points to the name of the stanza after which the new stanza is to be inserted. If this parameter is NULL, then the stanza is added to the end of the file.

All information that is repeated in the **default** stanza of the attribute file is removed from the new stanza before it is written to the file.

The calling program must have an effective user ID of superuser to access system customization files such as **/etc/master**, **/etc/system**, and **/etc/predefined**.

Return Value
Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgcadsz** subroutine fails, the following value is returned:

CFG_ECLS An error occurred while closing a file.

CFG_EOPN An error occurred while opening a file.

CFG_SPCE Memory allocation failed because of insufficient space.

CFG_UNIO An unrecoverable I/O error occurred during processing.

Related Information
In this book: "cfgadev" in topic 1.2.31, "cfgcclsf" in topic 1.2.34, "cfgcdlsz" in topic 1.2.35, "cfgcopsf" in topic 1.2.36, "cfgcrdsz" in topic 1.2.37, and "attributes" in topic 2.3.5.

1.2.34 cfgcclsf

Purpose
Closes an attribute file.

Library
Run-time Services Library (**librts.a**)

Syntax

#include <cfg04.h>

int cfgcclsf (*atfile***)**
CFG__SFT ***atfile;

Description
The **cfgcclsf** subroutine closes an attribute file. (For details about attribute files, see "attributes" in topic 2.3.5.)

The *atfile* parameter points to an open attribute file structure.

The calling program must have an effective user ID of superuser to access system customization files such as **/etc/master**, **/etc/system**, and **/etc/predefined**.

Return Value
Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgcclsf** subroutine fails, then the following value is returned:

CFG_UNIO Unrecoverable I/O error occurred during processing.

Related Information
In this book: "cfgcadsz" in topic 1.2.33, "cfgcdlsz" in topic 1.2.35, "cfgcopsf" in topic 1.2.36, "cfgcrdsz" in topic 1.2.37, and "attributes" in topic 2.3.5.

1.2.35 cfgcdlsz

Purpose
Deletes a stanza from an attribute file.

Library
Run-time Services Library (**librts.a**)

Syntax

```
#include <cfg04.h>

int cfgcdlsz (atfile, stname)
CFG__SFT *atfile;
char *stname;
```

Description
The **cfgcdlsz** subroutine deletes a stanza from an attribute file. (For details about attribute files, see "attributes" in topic 2.3.5.)

The *atfile* parameter points to an open attribute file structure. The *stname* parameter points to the name of the stanza to be deleted from the file.

The calling program must have an effective user ID of superuser to access system customization files such as **/etc/master**, **/etc/system**, and **/etc/predefined**.

Return Value
Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgcdlsz** subroutine fails, one of the following values is returned:

CFG_ECLS An error occurred while closing a file.

CFG_EOPN An error occurred while opening a file.

CFG_SPCE Memory allocation failed because of insufficient space.

CFG_SZBF The file contains a stanza that is larger than the maximum allowable stanza size.

CFG_SZNF The requested stanza to be deleted was not found in the file.

CFG_UNIO An unrecoverable I/O error occurred during processing.

Related Information
In this book: "cfgcadsz" in topic 1.2.33, "cfgcclsf" in topic 1.2.34, "cfgcopsf" in topic 1.2.36, "cfgcrdsz" in topic 1.2.37, "cfgddev" in topic 1.2.38, and "attributes" in topic 2.3.5.

1.2.36 cfgcopsf

Purpose
Opens an attribute file.

Library
Run-time Services Library (**librts.a**)

Syntax

```
#include <cfg04.h>

CFG__SFT *cfgcopsf (path)
char *path;
```

Syntax
The **cfgcopsf** subroutine opens an attribute file for update. (For details about attribute files, see "attributes" in topic 2.3.5.)

The **path** parameter points to the full path name of the file to be opened.

The **cfgcopsf** subroutine calls the **fopen** subroutine to open the file for update. If the call to **fopen** is successful, **cfgcopsf** allocates a **CFG__SFT** structure. This structure contains the file descriptor returned by **fopen**, a pointer to a default stanza buffer for reads, a pointer to an array of indexes in a default stanza buffer, and the full path name of the file that was opened.

The calling program must have an effective user ID of superuser to access system customization files such as **/etc/master**, **/etc/system**, and **/etc/predefined**.

Return Value
Upon successful completion, the **cfgcopsf** subroutine returns a pointer to an open attribute file structure. If the **cfgcopsf** subroutine fails, it returns a NULL pointer.

Related Information
In this book: "cfgcadsz" in topic 1.2.33, "cfgcclsf" in topic 1.2.34, "cfgcdlsz" in topic 1.2.35, "cfgcrdsz" in topic 1.2.37, "fopen, freopen, fdopen" in topic 1.2.82, and "attributes" in topic 2.3.5.

1.2.37 cfgcrdsz

Purpose
Reads an attribute file stanza.

Library
Run-time Services Library (**librts.a**)

Syntax

```
#include <cfg04.h>

int cfgcrdsz (atfile, stanza, nbytes, stname)
CFG__SFT *atfile;
char *stanza;
int nbytes;
char *stname;
```

Description
The **cfgcrdsz** subroutine reads one stanza from an attribute file. A specific stanza may be requested, or the next stanza in the file can be read. When a stanza is read, any information contained in a **default** stanza preceding it in the file is added to the information returned in the buffer. (For details about attribute files, see "attributes" in topic 2.3.5.)

The **atfile** parameter points to an open attribute file structure.

The **stanza** parameter points to the buffer into which the stanza will be read.

The **nbytes** parameter is the size in bytes of the buffer pointed to by the **stanza** parameter.

The **stname** parameter points to a string containing the name of the stanza to be read. If this parameter is a NULL pointer, then the next stanza in the file is read.

The calling program must have an effective user ID of superuser to access system customization files such as **/etc/master**, **/etc/system**, and **/etc/predefined**.

Return Value
Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgcrdsz** subroutine fails, then one of the following values is returned:

CFG_EOF The next stanza was requested, but the end of the file has been reached.

CFG_SZNF The requested stanza was not found in the file.

CFG_SZBF The requested stanza is longer than **nbytes** bytes.

CFG_UNIO Unrecoverable I/O error occurred during processing.

Related Information
In this book: "cfgcadsz" in topic 1.2.33, "cfgcclsf" in topic 1.2.34, "cfgcdlsz" in topic 1.2.35, "cfgcopsf" in topic 1.2.36, and "attributes" in topic 2.3.5.

1.2.38 cfgddev

Purpose
Deletes a device.

Library
Run-time Services Library (**librts.a**)

Syntax

```
#include <cfg01.h>

int cfgddev (master, system, dstname, vflag, cflag)
char *master, *system, *dstname;
int vflag;
int cflag;
```

Description
The **cfgddev** subroutine deletes information about devices and device drivers from the system configuration.

The **master** parameter points to the full path name of the **master** file. The **system** parameter points to the full path name of the **system** file. These files are usually **/etc/master** and **/etc/system**, respectively. The **dstname** parameter points to a string containing the name of the stanza in the **system** file of the device to be deleted.

The **vflag** parameter is either 1 (for yes) or 0 (for no). If the **vflag** parameter is 1, then **cfgddev** executes the **osconfig** command with the **-d dstname** flag. The **osconfig** command then processes the named stanza for driver deletion and produces a shell procedure. The **cfgddev** subroutine then runs this shell procedure to delete the special file (**/dev** file) for the device. If the **osconfig** command returns an error, then the device is not deleted.

The **cflag** parameter is either 1 (for no) or 0 (for yes), for **osconfig** to call customize helpers when deleting devices.

If the device stanza named by the **dstname** parameter contains the **specproc** keyword, then the program specified by the value of this keyword is executed to perform any special processing required when deleting this device. The value of the **specproc** keyword must be the full path name of an executable file. The following arguments are passed to the program using the **argv** mechanism described in "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71. All of them are passed as character strings.

argv[0] The full path name of the special-processing program
argv[1] The full path name of the **master** file
argv[2] The full path name of the **system** file
argv[3] The name of the device stanza
argv[4] The character string **"d"**, indicating deletion.

If the special processing program fails, then the device is still deleted from the system, but some additional steps may be required to clean up the system.

The device stanza associated with the deleted device is then deleted from the **system** file.

Return Value
Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgddev** subroutine fails then one of the following values is returned:

CFG_CLSE An error was detected while trying to close a file.

CFG_DVND The device could not be deleted from the **system** file.

CFG_DVNF The device to be deleted cannot be found in the **system** file.

CFG_FCOR The **master** or **system** file is set up incorrectly.

CFG_MALF Memory allocation failed because of insufficient space.

CFG_OPNE An error was detected while trying to open a file.

CFG_SLPF Special processing failed. The device is deleted, but some additional steps may be required to clean up the system.

CFG_OCFG The **osconfig** command failed.

CFG_REBOOT AIX must be rebooted by calling the **cfgaply** routine, although **cfgddev** was successful. **cfgddev** will return **CFG_REBOOT** if the last stanza in **/etc/system** associated with a AIX device driver is deleted and the mandatory attribute in the driver's **/etc/master** stanza is set to false.

CFG_RBSLPF Special processing failed and a AIX reboot must be performed.

Related Information
In this book: "attributes" in topic 2.3.5, "master" in topic 2.3.32, and "system" in topic 2.3.56.

The **osconfig** command in *AIX Operating System Commands Reference*.

1.2.39 cfgetospeed, cfsetospeed, cfgetispeed, cfsetispeed

Purpose
Get and set the input and output baud rates.

Library
Standard I/O Library (**libc.a**)

Syntax

```
#include <termios.h>

speed_t cfgetospeed(termios_p)
struct termios *termios_p;

speed_t cfsetospeed(termios_p, speed)
struct termios *termios_p;
speed_t speed;

speed_t cfgetispeed(termios_p)
struct termios *termios_p;

speed_t cfsetispeed(termios_p, speed)
struct termios *termios_p;
speed_t speed;
```

Description
cfgetospeed returns the output baud rate store in the **termios** structure to which **termios_p** points.

cfsetospeed sets the output baud rate stored in the **termios** structure to which **termios_p** points to **speed**.

cfgetispeed returns the input baud rate stored in the **termios** structure.

cfsetispeed sets the input baud rate stored in the **termios** structure to **speed**.

Return Value
Both **cfsetispeed** and **cfsetospeed** return 0 if successful and -1 to indicate an error.

Error Conditions
The **cfsetispeed** and **cfsetospeed** subroutines will fail if the following is true:

EINVAL The speed is not a valid baud rate.

Related Information
In this book, "tcgetattr, tcsetattr" in topic 1.2.299, "tcsendbreak, tcdrain, tcflush, tcflow" in topic 1.2.301, and "tcgetpgrp, tcsetpgrp" in topic 1.2.300.

1.2.40 chdir

Purpose
Changes the current directory.

Syntax

int chdir (path)
char *path;

Description
The **chdir** system call changes the current directory to the directory specified by the **path** parameter. The ***current directory***, also called the ***current working directory***, is the starting point of searches for path names that do not begin with a **/** (slash).

Warning: After changing into a directory that uses a symbolic link, attempts to traverse the tree using "**..**" may produce unexpected results. "**..**" refers to the hard link parent of the directory, obtained by deleting the last element from *path*, not the symbolic link parent.

Note: If the working directory is a directory stored only on sites which are no longer available, it is not possible to use **chdir** to change into its subdirectories and using **chdir ..** to change to the parent directory will only work if the parent directory is stored on an available site.

Return Value
Upon successful completion, the **chdir** system call returns a value of 0. If the **chdir** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **chdir** system call fails and the current directory remains unchanged if one or more of the following are true:

ENOTDIR A component of the **path** parameter is not a directory.

ENOENT The named directory does not exist.

EACCES Search permission is denied for any component of the **path** parameter.

EFAULT The **path** parameter points to a location outside of the process's allocated address space.

ESTALE The process's root or current directory is located in an NFS virtual file system that has been unmounted.

ENAMETOOLONG
A component of the **path** parameter exceeded **NAME_MAX** characters or the entire **path** parameter exceeded **PATH_MAX** characters.

ENOENT A hidden directory was named, but no component inside it matched the process's current site path list.

ENOENT A symbolic link was named, but the file to which is refers does not exist.

ELOOP A loop of symbolic links was detected.

If the Transparent Computing Facility is installed on your system, **chdir** can also fail if one or more of the following are true:

ESITEDN1 The **path** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE The **path** is a name relative to the current directory, but no site which stores this directory is currently up.

ENOSTORE A component of **path** is replicated but is not stored on any site which is currently up.

EINTR A signal was caught during the **chdir** system call.

Related Information
In this book: "chroot" in topic 1.2.46.

The **cd** command in *AIX Operating System Commands Reference*.

1.2.41 chfstore

Purpose
Changes replicated storage attribute of a file.

Syntax

```
#include <sys/types.h>

int chfstore (path, fstore)
char *path;
fstore_t fstore;
```

Description
The **chfstore** system call sets the replicated storage attribute of the file named by **path** to the bit pattern contained in **fstore**. If the file does not reside on a replicated file system, **chfstore** has no effect and does not return an error.

The **fstore** bits are used to determine whether a copy of a file is stored in a nonprimary, nonbackbone copy of a replicated file system. Each copy of a replicated file system contains an **fstore** field in the super block. The system stores the file on those copies where the bitwise AND of the file's **fstore** field and the copy's **fstore** field is nonzero.

If the last component of **path** is a symbolic link, **chfstore** changes the **fstore** bits on the symbolic link rather than the file pointed to by the symbolic link.

The effective user ID of the process must match the owner of the file or be superuser to change the **fstore** of a file.

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **chfstore** system call fails and the replicated storage attribute is unchanged if one or more of the following are true:

EPERM The effective user ID does not match the owner of the file and the effective user ID is not superuser.

EROFS The named file resides in a replicated file system in which the primary copy is unavailable, or in a replicated file system which has been mounted read-only.

ENOTDIR A component of the path prefix is not a directory.

ENOENT The named file does not exist.

ENOENT A null path name is provided.

ENOENT A hidden directory is named, but no component inside it matches the process's current site path list.

ENOENT A symbolic link is named, but the file to which it refers does not exist. (Since **chfstore** does not follow a symbolic link when it is the last component of the path, this error cannot occur on

the last component).

EACCES Search permission is denied on a component of the path prefix.

EFAULT **path** points outside the process's allocated address space.

If the Transparent Computing Facility is installed on your system **chfstore** could fail if the following is true:

EINTR A signal was caught during the **chfstore** system call. This can occur if the internal open of this file is suspended because of a topology change.

ESITEDN1 **path** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE **path** is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of **path** is replicated but is not stored on any site which is currently up.

ELOOP A loop of symbolic links was detected. Since **chfstore** does not follow a symbolic link when it is the last component of the path, this error cannot occur on the last component.

Related Information
In this book: "fs" in topic 2.3.20.

The **chfstore** and **store** commands in *AIX Operating System Commands Reference*.

1.2.42 chhidden

Purpose
Changes the hidden attribute of a directory.

Syntax

```
int chhidden(dirname,hideflag)
char *dirname;
int hideflag;
```

Description
The **chhidden** system call allows the superuser to turn a normal directory into a hidden directory or vice versa. If **hideflag** is not zero, the directory pointed to by **dirname** is converted to a hidden directory. If it is zero, the directory is converted to a normal directory.

When using **chhidden** to reference a directory which is already a hidden directory it is unnecessary for **dirname** to explicitly name the hidden directory (using the @ notation) as **chhidden** does not expand hidden directory references.

The **chhidden** system call does not enforce the restriction that only regular and special files can appear in a hidden directory (see "Hidden Directories" in topic 1.1.5.1.5). Applications which use this call are required to make the necessary checks and enforce this rule.

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
Possible errors:

EPERM The calling process lacks superuser privileges.

ENOTDIR **dirname** does not name a directory.

EROFS The named directory resides on a read-only file system.

ENOTDIR A component of the path prefix of **dirname** is not a directory.

ENOENT The named file does not exist.

ENOENT A null path name was provided.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

EACCES Search permission is denied on a component of the path prefix of **dirname**.

EFAULT **dirname** points outside the process's allocated address space.

ELOOP A loop of symbolic links was detected.

If the Transparent Computing Facility is installed on your system, **chhidden** can also fail if one or more of the following are true:

ESITEDN1 **dirname** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE **dirname** is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of **dirname** is replicated but is not stored on any site which is currently up.

EROFS Write access is requested for a file on a replicated file system in which the primary copy is unavailable.

EINTR A signal was caught during the system call.

Examples

In the example below an @ is given at the end of the directory name. This is necessary for the **statx** system call but not for **chhidden**, since **chhidden** does not pass through hidden directories.

```
 /* This program changes a hidden directory into a regular
directory or changes a regular directory into a hidden directory. */

#include <sys/types.h>
#include <sys/stat.h>
main()
{

     char fname[50];
     struct stat stat;

     strcpy(fname,"/tmp/hidtest@");
     if (statx(fname, &stat, sizeof(struct stat), STX_HIDDEN) != 0) {
          printf("bad statx\n");
          exit(1);
     }
     if (S_ISHIDDEN (stat.st_mode))
          chhidden(fname,0);
     else
          chhidden(fname,1);
}
```

Related Information
In this book: "stat.h" in topic 2.4.22 and "mkdir" in topic 1.2.168.

1.2.43 chlwm

Purpose
Updates the low-water mark of a file system.

Syntax

```
#include <sys/types.h>

commitcnt_t chlwm(gfs, site, lwm)
gfs_t gfs;
siteno_t site;
commitcnt_t lwm;
```

Description
The **chlwm** system call sets the low-water mark of the local site's copy of the file system identified by **gfs** to the greater of **lwm** and the current file system's low-water mark. It returns the previous low-water mark.

If **site** is not the local site, the low-water mark is returned, but the remote file system is not changed.

The low-water mark is a field in the superblock of a replicated file system copy which records which changes have already been propagated to this copy from the primary copy. It is normally updated by the AIX kernel as changes made to the primary copy are automatically propagated to the copy. When propagation is done outside the kernel by the file system reconciliation procedure (**primrec**), the final step in this process is to adjust the low-water mark with this call. Other programs are not expected to use this system call. For more information on replicated file systems see "Distributed File Systems Overview" in *AIX/370 Administration Guide*.

You must have superuser authority to call **chwlm()**.

Error Conditions
If **chlwm** fails, it returns a value of -1 and sets the following error codes:

EPERM The user was not superuser.

EBADST The **site** was 0.

ENOSTORE The specified file system is not a replicated file system.

ENOSTORE The specified file system was not mounted.

EINVAL The specified file system was not mounted at **site** or is not a replicated file system.

ESITEDN1 The **site** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

Related Information
In this book: "spropin" in topic 1.2.279, "raccept" in topic 1.2.219, and "fs" in topic 2.3.20.

The **primrec** and **recmstr** commands in *AIX Operating System Commands Reference*.

1.2.44 chmod, fchmod

Purpose
Changes file access permissions.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/stat.h>


int chmod (path, mode)          int fchmod (fd, mode)
char *path;                     int fd, mode;
int mode;
```

Description
The **chmod** system call sets the access permissions of the file specified by the **path** parameter.

The **fchmod** system call sets the access permissions on the **open** file specified by the file descriptor parameter **fd**. If **path** names a symbolic link, **chmod** sets the access permissions on the file to which the symbolic link points. If **path** names a hidden directory, **chmod** sets the access permission on a selected file within the hidden directory. The access permissions of the file are set according to the bit pattern specified by the **mode** parameter.

To change file access permissions, the effective user ID of the calling process must either be superuser or match the ID of the file's owner.

The **mode** parameter is constructed by logically ORing one or more of the following values, which are defined in the **sys/stat.h** header file:

S_ISUID	Sets the process's effective user ID to the file's owner on execution.
S_ISGID	Sets the process's effective group ID to the file's group on execution.
S_ISVTX	Saves text image after execution.
S_ENFMT	Enables enforcement-mode record locking.
S_IRUSR	Permits the file's owner to read it.
S_IWUSR	Permits the file's owner to write to it.
S_IXUSR	Permits the file's owner to execute it (or to search the directory).
S_IRGRP	Permits the file's group to read it.
S_IWGRP	Permits the file's group to write to it.
S_IXGRP	Permits the file's group to execute it (or to search the directory).
S_IROTH	Permits others to read the file.
S_IWOTH	Permits others to write to the file.
S_IXOTH	Permits others to execute the file (or to search the directory).

Other mode values exist that can be set with the **mknod** system call, but not with **chmod** or **fchmod**. A complete list of the possible file mode values and other useful macros appears in "stat.h" in topic 2.4.22.

Setting **S_ISVTX** on a shared executable file prevents the system from unmapping the program text segment of the file when its last user

terminates. Thus, when the next process executes it, the text need not be read from the file system. It is simply paged in, saving time. The calling process must have superuser authority to set the **S_ISVTX** mode bit on a regular file.

Setting **S_ISVTX** on a directory marks that directory such that only the following users may remove files from the directory:

1. the owner of the directory, or

2. the owner of the files, or

3. a process with superuser authority.

Setting **S_ISVTX** on a character special file marks that file as a ***multiplex special file***. Multiplex special files are special files which appear to the system to be a directory full of special files and are used to implement virtual devices such as **/dev/hft**. See "hft" in topic 2.5.11.

Setting **S_ENFMT** (which has the same value as **S_ISGID**) on a regular file, while setting no execute permission bits, marks the file such that all file locks placed on the file with **fcntl, lockf** or **flock** are treated as enforced record locks (see "fcntl.h" in topic 2.4.8). It is undefined what the behavior is if the **S_ENFMT** mode bit is changed on a file which has existing record locks. Attempts to do this in subsequent releases of AIX may result in an error being returned.

Setting **S_ISGID** on a directory marks the directory in **BSD** compatibility mode, causing files and directories subsequently created within the directory to inherit their group IDs from this directory. Otherwise, newly created files and directories inherit their group IDs from the effective group ID of the process which created them.

Setting **S_ISGID** on a regular file along with one of the three execute permission bits enables the set-group-ID behavior of **exec**. This is permitted only if the calling process has superuser authority or if one of the IDs in the calling process's group access list matches the group ID file. Otherwise, the **S_ISGID** bit is cleared (see "getgroups" in topic 1.2.97 or "setgroups" in topic 1.2.249 for more information about group access lists).

Return Value
Upon successful completion, the **chmod** and **fchmod** system calls return a value of 0. If the **chmod** or **fchmod** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **chmod** system call fails and the file permissions remain unchanged if one or more of the following is true:

ENOTDIR A component of the **path** parameter is not a directory.

ENOENT The named file does not exist.

EACCES A component of the **path** parameter has search permission denied.

EFAULT The **path** parameter points to a location outside of the process's allocated address space.

ENAMETOOLONG

A component of the **path** parameter exceeds **NAME_MAX** characters or the entire **path** parameter exceeds **PATH_MAX** characters.

ENOENT A hidden directory is named, but no component inside it matches the process's current site path list.

ENOENT A symbolic link is named, but the file to which it refers does not exist.

ELOOP A loop of symbolic links is detected.

ENOSTORE The **path** is a name relative to the working directory, but no site which stores this directory is currently up.

EINTR A signal is caught during the **chmod** system call.

The **fchmod** system call fails and file permissions remain unchanged if one or more of the following are true:

EBADF The descriptor is not valid.

EINVAL **fd** refers to a socket, not to a file.

The **chmod** and **fchmod** system calls fail and file permissions remain unchanged if one or more of the following are true:

EPERM The effective user ID does not match the ID of the owner of the file or the ID of superuser.

EROFS The named file resides on a read-only file system.

ESTALE The process's root or current directory is located in an NFS virtual file system that has been unmounted.

If the Transparent Computing Facility is installed on your system, **chmod** can also fail if one or more of the following are true:

ESITEDN1 The specified file cannot be accessed because a site failed.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE The file specified by **fd** or a component of the **path** is replicated but is not stored on any site which is currently up.

EROFS Write access is requested for a file on a replicated file system in which the primary copy is unavailable.

EIO An I/O error occurs while reading from or writing to a file system.

Related Information
In this book: "chown, fchown" in topic 1.2.45, "getgroups" in topic 1.2.97, "mknod, mknodx, mkfifo" in topic 1.2.169, and "setgroups" in topic 1.2.249.

The **chmod** command in *AIX Operating System Commands Reference*.

1.2.45 chown, fchown

Purpose
Changes the owner and group IDs of files associated with a path name or with a file descriptor.

Library
Standard C Library (**libc.a**)

Syntax

```
                                        int fchown  (fd, owner, group)
int chown  (path, owner, groint fd;
char *path;                             uid_t owner;
uid_t owner;                            gid_t group;
gid_tgroup;
```

Description
The **chown** and **fchown** system calls change the owner ID and/or the group ID of the file named by the **path** parameter or of the open file named by the **fd** parameter, respectively. If the named file is a symbolic link, the owner ID and/or group ID of the symbolic link itself are changed, not those of the file pointed to by the symbolic link.

The owner and group IDs of the named file are set to the numeric values contained in the **owner** and **group** parameters, respectively.

If either the **owner** or **group** parameter has the value **(uid_t)-1** or **(gid_t)-1**, respectively the corresponding ID in the named file is left unchanged.

A process can change the ownership of a file only if its effective user ID is the superuser. The group ID may be changed if the effective user ID of the process is superuser or matches the file's owner ID. In the latter case, the group ID may only be changed to the value of the process's effective group ID or a value in the process's concurrent group list.

Return Value
Upon successful completion, a value of 0 is returned. If the **chown** or **fchown** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **chown** system call fails and the owner ID and the group ID of the named file remain unchanged if one or more of the following are true:

ENOTDIR A component of the *path* prefix is not a directory.

ENOENT The named file does not exist.

EACCES Search permission is denied on a component of the *path* prefix.

ENOENT A symbolic link was named, but the file to which it refers does exist. Since **chown** does not follow a symbolic link when it is the last component of the *path*, this error cannot occur on the last component.

EFAULT The **path** parameter points to a location outside of the process's allocated address space in an NFS virtual file system that has been unmounted.

ENAMETOOLONG
A component of the **path** parameter exceeded **NAME_MAX** characters or the entire **path** parameter exceeded **PATH_MAX** characters.

ENOENT A hidden directory was named, but no component inside it matched the process's current site path list.

ELOOP A loop of symbolic links was detected. Since **chown** does not follow symbolic links in the last component of the path, this error cannot occur on the last component.

ENOSTORE The **path** is a name relative to the working directory, but no site which stores this directory is currently up.

EINTR A signal was caught during the system call.

The **fchown** system call fails and the owner ID and group ID of the named file remain unchanged if one or more of the following are true:

EBADF **fd** does not refer to a valid file descriptor

EINVAL **fd** refers to a socket, not a file.

The **chown** and **fchown** system calls fail and the owner ID and group ID of the named file remain unchanged if one or more of the following are true:

EPERM The effective user ID does not match the owner of the file and the effective user ID is not superuser.

EROFS The named file resides on a read-only file system.

ESTALE The process's root or current directory is located

If the Transparent Computing Facility is installed on your system, **chown** can also fail if one or more of the following are true:

ESITEDN1 The specified file cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE The file specified by **fd** or a component of **path** is replicated but is not stored on any site which is currently up.

EROFS Write access is requested for a file on a replicated file system in which the primary copy is unavailable.

EIO An I/O error occurred while reading from or writing to the file system.

Related Information
In this book: "chmod, fchmod" in topic 1.2.44, "statx, fstatx, stat, fstat, fullstat, ffullstat, lstat" in topic 1.2.282, and "stat.h" in topic 2.4.22.

The **chown** command in *AIX Operating System Commands Reference*.

1.2.46 chroot

Purpose
Changes the effective root directory.

Syntax

```
int chroot (path)
char *path;
```

Description
The **chroot** system call causes the directory named by the **path** parameter to become the effective root directory. The ***effective root directory*** is the starting point when searching for a file whose path name begins with **/** (slash). The current directory is not affected by the **chroot** system call.

The effective user ID of the calling process must be superuser to change the effective root directory.

The **..** (dot-dot) entry in the effective root directory is interpreted to mean the effective root directory itself. Thus, **..** (dot-dot) cannot be used to access files outside the subtree rooted at the effective root directory.

The **chroot** system call simulates a call to **setlocal** with the process's current **<LOCAL>** alias, and as a result, the alias is re-evaluated within the new root directory. If the new root directory does not contain a directory named by the process's current **<LOCAL>** alias, subsequent references to symbolic links which begin with "**<LOCAL>**" return an ENOENT error.

Return Value
Upon successful completion, a value of 0 is returned. If the **chroot** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **chroot** system call fails and the effective root directory remains unchanged if one or more of the following are true:

ENOTDIR Any component of the path name is not a directory.

ENOENT The named directory does not exist.

EPERM The effective user ID of the calling process is not superuser.

EFAULT The **path** parameter points to a location outside of the process's allocated address space.

ESTALE The process's root or current directory is located in an NFS virtual file system that has been unmounted.

ENAMETOOLONG
A component of the **path** parameter exceeded **NAME_MAX** characters, or the entire **path** parameter exceeded **PATH_MAX** characters.

ENOENT A hidden directory was named, but no component inside it matched the process's current site path list.

ENOENT A symbolic link was named, but the file to which it refers does

not exist.

ENOENT A NULL path name was provided.

ELOOP A loop of symbolic links was detected.

If the Transparent Computing Facility is installed on your system, **chroot** can also fail if one or more of the following are true:

ESITEDN1 The **path** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE The **path** is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of **path** is replicated but not stored on any site which is currently up.

EINTR A signal was caught during the **chroot** system call.

Related Information
In this book: "chdir" in topic 1.2.40, "getlocal, setlocal" in topic 1.2.102.

The **chroot** command in *AIX Operating System Commands Reference*.

1.2.47 clock

Purpose
Reports CPU time used.

Library
Standard C Library (**libc.a**)

Syntax

long clock ( )

Description
The **clock** subroutine returns the amount of CPU time (in microseconds) used since the first call to **clock**.

The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed a **wait** system call or a **system** subroutine.

Note: The value returned by the **clock** subroutine is defined in microseconds for compatibility with systems that have CPU clocks with very high resolution. Because of this, the value returned wraps around after accumulating approximately 2147 seconds of CPU time (about 36 minutes).

Related Information
In this book: "system" in topic 1.2.298, "times" in topic 1.2.304, and "wait, waitpid" in topic 1.2.325.

1.2.48 close, closex

Purpose
Closes the file associated with a file descriptor.

Syntax

```
int close (fildes)
int fildes;

int closex (fildes, ext)
int fildes, ext;
```

Description
The **close** and **closex** system calls close the file associated with the file descriptor **fildes**.

The **fildes** parameter is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call. The **ext** parameter provides communication with character device drivers that require additional information or return additional status. Each driver interprets the **ext** parameter in a device-dependent way, either as a value or as a pointer to a communication area. Drivers must apply reasonable defaults when the **ext** parameter is 0.

If the file is open in **O_DEFERC** mode, changes are made permanent by using the commit mechanism at some point after the file is closed. The commit operation implicit in **close** is done asynchronously, and consequently, problems such as I/O errors and losses of file servers may go unreported. In these situations, the content of the file is rolled back to the previously committed version (see "fabort" in topic 1.2.75).

Programs which desire synchronous error reporting and a guarantee that the changes to the file have been made permanent should use the **fcommit** system call explicitly before using **close**.

If the file is open for writing using more than one file descriptor (by this process or in conjunction with other processes) the **commit** semantics described above apply only if the **O_DEFERC** flag is set on for all opens. Any other **open** or **creat** system call may cause the file contents to be permanently updated periodically (see "sync" in topic 1.2.295).

All locks held on the file by the calling process, whether they are applied using this file descriptor or another one, are released when the file is closed (see "fcntl, flock, lockf" in topic 1.2.78). If the **fildes** parameter is associated with a mapped file and if no other process has attached this mapped file, it is unmapped.

Return Value
Upon successful completion, a value of 0 is returned. If the **close** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **close** and **closex** system calls fail if the following is true:

EBADF The **fildes** parameter is not a valid open file descriptor.

Related Information
In this book: "open, openx, creat" in topic 1.2.199, "dup" in topic 1.2.64, "exec: execl, execv, execle, execve, execlp, execvp" in

topic 1.2.71, "fcntl, flock, lockf" in topic 1.2.78, "pipe" in topic 1.2.204, and "socket" in topic 1.2.275.

1.2.49 connect

Purpose
Initiates a connection on a socket.

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>

int connect (s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

Description
The parameter **s** is a socket. If it is of type **SOCK_DGRAM**, this system call specifies the peer with which the socket is to be associated. Datagrams are to be sent to this address, and it is the only address from which datagrams are to be received.

If the socket is of type **SOCK_STREAM**, this system call attempts to make a connection to another socket. The other socket is specified by **name**, which is an address in the communication space of the socket. Each communication space interprets the **name** parameter in its own way.

Generally, stream sockets may successfully connect only once; datagram sockets may use **connect** multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

Return Value

Upon successful completion, a value of 0 is returned. If the **connect** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **connect** system call fails if one or more of the following are true:

EBADF The **s** parameter is not valid.

ENOTSOCK The **s** parameter refers to a file, not a socket.

EADDRNOTAVAIL The specified address is not available from the local machine.

EAFNOSUPPORT The addresses in the specified address family cannot be used with this socket.

EISCONN The socket is already connected.

ETIMEDOUT The establishment of a connection timed out before a connection was made.

ECONNREFUSED The attempt to connect was rejected.

ENETUNREACH The network is not reachable from this host.

EADDRINUSE The specified address is already in use.

EFAULT The **addr** parameter is not in a writable part of the user address space.

EINPROGRESS The socket is marked nonblocking and the connection cannot be completed immediately. Using the **select** system call, completion can be determined by selecting the socket for writing.

EALREADY The socket is marked nonblocking and a previous connection attempt has not yet completed.

The following errors are specific to connecting names in the **AF_UNIX** address family.

ENAMETOOLONG
A component of the path name exceeded **NAME_MAX** characters, or the entire path name exceeded **PATH_MAX** characters.

ENOTDIR A component of the path name is not a directory.

EINVAL The path name contains a character with the high-order bit set.

ENOENT The named socket does not exist.

ELOOP A loop of symbolic links was detected.

EACCES Search permission is denied for a component of the path prefix.

EACCES Write access to the named socket is denied.

Related Information
In this book: "accept" in topic 1.2.9, "getsockname" in topic 1.2.120, "select" in topic 1.2.242, and "socket" in topic 1.2.275.

1.2.50 conv

Purpose
Translates characters.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <ctype.h>
#include <NLctype.h>
```

```
int toupper (c)                 int NCtoupper (x)
int c;                          int x;

int tolower (c)                 int NCtolower (x)
int c;                          int x;

int _toupper (c)                int _NCtoupper (x)
int c;                          int x;

int _tolower (c)                int _NCtolower (x)
int c;                          int x;

int toascii (c)                 int NCtoNLchar (x)
int c;                          int x;

int NCesc (xp, cp)              int NCunesc (cp, xp)
NLchar *xp;                     char *cp;
char *cp;                       NLchar *xp;

wchar_t towlower (c)            int NCflatchr (x)
wchar_t c;                      int x;
wchar_t towupper (c)
wchar_t c;

wchar_t towascii (c)
wchar_t c;
```

Description
The **NCxxxxxxx** subroutines translate all characters, including extended characters, as code points (see "Introduction to International Character Support" in *Managing the AIX Operating System*). The other subroutines translate traditional ASCII characters only.

The **toupper** and the **tolower** subroutines have as domain the range of the **getc** subroutine: from -1 through 255.

If the parameter of the **toupper** subroutine represents a lowercase letter, the result is the corresponding uppercase letter. If the parameter of the **tolower** subroutine represents an uppercase letter, the result is the corresponding lowercase letter. All other values in the domain are returned unchanged.

The **_toupper** and **_tolower** routines are macros that accomplish the same thing as **toupper** and **tolower**, but they have restricted domains and they are faster. **_toupper** requires a lowercase letter as its parameter; its

result is the corresponding uppercase letter. **_tolower** requires an uppercase letter as its parameter; its result is the corresponding lowercase letter. Values outside the domain cause undefined results.

The value of **x** is in the domain of any legal **NLchar** in a value range from 0 to **NLCHARMAX** inclusive, or a special value of -1 (which represents EOF).

If the parameter of the **NCtoupper** subroutine represents a lowercase letter according to the current collating sequence configuration, the result is the corresponding uppercase letter. If the parameter of the **NLtolower** subroutine represents an uppercase letter according to the current collating sequence configuration, the result is the corresponding lowercase letter. All other values in the domain are returned unchanged.

The **_NCtoupper** and **_NCtolower** routines are macros that accomplish the same thing as **NCtoupper** and **NCtolower**, but have restricted domains and are faster. **_NCtoupper** requires a lowercase letter as its parameter; its result is the corresponding uppercase letter. **_NCtolower** requires an uppercase letter as its parameter; its result is the corresponding lowercase letter. Values outside the domain cause undefined results.

The **toascii** macro yields the value of its parameter with all bits that are not part of a standard ASCII character turned off. It is intended for compatibility with other systems.

The **NCtoNLchar** macro yields the value of its parameter with all bits turned off that are not part of an **NLchar**.

The **NCesc** macro converts the **NLchar** value **xp** into one or more ASCII bytes stored in the character array pointed to by **cp**. If the **NLchar** represents an extended character, it is converted into a printable ASCII escape sequence that uniquely identifies the extended character. **NCesc** returns the number of bytes it wrote. See "display symbols" in topic 2.4.4 for a list that shows the escape sequence for each character.

The inverse conversion is performed by the **NCunesc** macro, translating an ordinary ASCII byte or escape sequence starting at **cp** into a single **NLchar** at **xp**. **NCunesc** returns the number of bytes it read.

The **NCflatchr** macro converts its parameter value into the single ASCII byte that most closely resembles the parameter character in appearance. If no ASCII equivalent exists, it converts the parameter value to a **?** (question mark).

Note: In the multibyte environment, the **NCesc**, **NCunesc**, and **NCflatchr** subroutines are provided for backward compatibility and support code page pc850 only.

If the parameter of the **towupper** subroutine is a wide lower case character, the result is the corresponding upper case wide character. If the parameter of the **towlower** subroutine is a wide upper case character, the result is the corresponding lower case wide character. All other parameters passes are returned unchanged.

Related Information
In this book: "ctype" in topic 1.2.55, "getc, fgetc, getchar, getw, getwc, fgetwc, getwchar" in topic 1.2.91, and "display symbols" in topic 2.4.4.

"Introduction to International Character Support" in *Managing the AIX*

Operating System.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.51 copysign

Purpose
Copies the sign of a double-precision floating-point number.

Library
Math Library (**libm.a**)

Syntax

```
#include <math.h>

double copysign (x, y)
double x, y;
```

Description
The **copysign** subroutine copies the sign of one double-precision floating-point number to another. The **x** parameter contains the number to be changed to the sign of **y** parameter.

Return Value
The **copysign** subroutine returns **x** with the same sign as **y**.

1.2.52 crypt, encrypt, setkey

Purpose
Encrypts user passwords.

Library
Standard C Library (**libc.a**)

Syntax

```
                                        void encrypt (block, edflag)
char *crypt (key, salt)                 char *block;
char *key, *salt;                       int edflag;
void setkey (key)
char *key;
```

Description
The **crypt** and **encrypt** subroutines encrypt user passwords. They are based on a hashing encryption algorithm with variations intended to frustrate the use of hardware-implemented key searches. These subroutines are provided for compatibility with UNIX system implementations, and no assertion is made about the strength of the algorithm.

The **key** parameter is a user's typed password. The **salt** parameter is a two-character string chosen from the set [**a-zA-Z0-9./**].

The **salt** parameter is used to perturb the hashing algorithm in one of 4096 different ways, after which the password is used as the key to repeatedly encrypt a constant string. The return value points to the encrypted password. The first two characters of the return value are the string entered in the **salt** parameter.

The **crypt** subroutine uses a character array of length 64 containing only the values **(char) 0** and **(char) 1**. This string is divided into groups of eight characters each, and the low-order bit in each group is ignored. This provides a 56-bit key, which is set into the machine by **crypt**.

The other subroutines provide a somewhat primitive access to the actual hashing algorithm.

The **key** parameter to **setkey** is a character array of length 64, containing only the characters with numerical value 0 and 1. If this string is divided into groups of eight, the low-order bit in each group is ignored, leading to a 56-bit key which is set into the machine.

The **block** parameter to the **encrypt** subroutine is also a 64-character array containing only the values (**char**) 0 and (**char**) 1. **encrypt** modifies this array in place, producing a similar array that has been subjected to the hashing algorithm using the **key** set by **crypt** or **setkey**. If the **edflag** parameter is 0, the argument is encrypted; if nonzero, it is decrypted.

Note: Depending on license agreements, the setkey function and the decrypt capability of **encrypt** may be disabled.

Return Value
The **crypt** subroutine returns a pointer to the encrypted password. The first two characters of it are the same as the **salt** parameter.

Note: The return value points to static data that is overwritten by subsequent calls.

Error Conditions
The **crypt**, **encrypt**, and **setkey** subroutines fail if the following is true:

ENOSYS This functionality is not supported in this implementation.

Related Information
In this book: "getpass" in topic 1.2.108 and "passwd" in topic 2.3.44.

The **login** and **passwd** commands in *AIX Operating System Commands Reference*.

1.2.53 ctermid

Purpose
Generates a file name for terminal.

Library
Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>

char *ctermid (s)
char *s;
```

Description
The **ctermid** subroutine generates the path name of the controlling terminal for the current process and stores it in a string.

If the **s** parameter is a NULL pointer, the string is stored in an internal static area and the address is returned. The next call to **ctermid** overwrites the contents of the internal static area.

If the **s** parameter is not a NULL pointer, it points to a character array of at least **L_ctermid** elements as defined in the **stdio.h** header file. The path name is placed in this array and the value of **s** is returned.

The difference between the **ctermid** and **ttyname** subroutines is that **ttyname** must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while **ctermid** returns a string (**/dev/tty**) that refers to the terminal if used as a file name. Thus **ttyname** is useful only if the process already has at least one file open to a terminal.

Related Information
In this book: " ttyname, isatty, fullttyname" in topic 1.2.310.

1.2.54 ctime, localtime, gmtime, asctime, tzset

Purpose
Converts date and time to string representation.

Library
Standard C Library (**libc.a**)

Syntax
#include <time.h>

```
char *ctime (clock)          char *asctime (tm)
long *clock;                 struct tm *tm;

struct tm *localtime (clock)void tzset ( )
long *clock;
                             extern long timezone;
struct tm *gmtime (clock)    extern int daylight;
long *clock;                 extern char *tzname[2];
```

Description
The **ctime** subroutine converts a time value pointed to by the **clock** parameter, which represents the time in seconds since 00:00:00 Greenwich Mean Time (GMT), January 1, 1970, into a 26-character string in the following form:

```
Sun Sep 16 01:03:52 1973\n\0
```

The width of each field is always the same as shown here.

The **localtime** subroutine converts the long integer pointed to by the **clock** parameter, which contains the time in seconds since 00:00:00 GMT, January 1, 1970, into a **tm** structure. **localtime** adjusts for the time zone and for daylight saving time, if it is in effect.

The **gmtime** subroutine converts the long integer pointed to by the **clock** parameter into a **tm** structure containing the Greenwich Mean Time, which is the time that AIX uses. The **gmtime** routine always sets *tm_isdst* to FALSE; *localtime* must be used to inquire about daylight savings time.

The **tm** structure is defined in the **time.h** header file, and it contains the following members:

```
int tm_sec;   /* Seconds (0 - 59) */
int tm_min;   /* Minutes (0 - 59) */
int tm_hour;  /* Hours (0 - 23) */
int tm_mday;  /* Day of month (1 - 31) */
int tm_mon;   /* Month of year (0 - 11) */
int tm_year;  /* Year - 1900 */
int tm_wday;  /* Day of week (Sunday = 0) */
int tm_yday;  /* Day of year (0 - 365) */
int tm_isdst; /* Nonzero = Daylight saving time */
```

The **asctime** subroutine converts a **tm** structure to a 26-character string of the same format as **ctime**.

If the **TZ** environment variable is defined, its value overrides the default time zone, which is the U.S. Eastern time zone. See "environment" in topic 2.4.6 for the format of the time zone information specified by **TZ**.

TZ is usually set when the system is started up. Its value is defined in either **/etc/environment** or **/etc/profile**. It can also be set by the user as a regular environment variable for performing alternate time zone conversions.

The **tzset** subroutine sets the **timezone**, **daylight**, and **tzname** external variables to reflect the setting of **TZ**. **tzset** is called by **ctime** and **localtime**, and it can also be called explicitly by an application program.

The **timezone** external variable contains the difference, in seconds, between GMT and local standard time. For example, **timezone** is 5 ¦ 60 ¦ 60 for U.S. Eastern Standard Time.

The **daylight** external variable is nonzero if an alternate timezone is defined in the TZ environment variable. By default, this conversion follows the standard U.S. conventions; other conventions can be specified. The default conversion algorithm adjusts for the peculiarities of U.S. daylight saving time in 1974 and 1975. See "environment" in topic 2.4.6 for information about specifying alternate daylight saving time conventions.

The **tzname** external variable contains the name of the standard time zone (**tzname[0**]) and of the time zone when daylight saving time is in effect (**tzname[1**]). For example:

```
char *tzname[2] = {"EST", "EDT"};
```

The **time.h** header file contains declarations of all these subroutines, externals, and the **tm** structure.

Warning: The return values point to static data that is overwritten by each call.

Related Information
In this book: "time" in topic 1.2.303, "getenv, NLgetenv" in topic 1.2.94, "NLstrtime" in topic 1.2.194, "NLtmtime" in topic 1.2.195, "profile" in topic 2.3.48, and "environment" in topic 2.4.6.

1.2.55 ctype

Purpose
Classifies characters.

Library
Standard C Library (**libc.a**)

Syntax

#include <ctype.h>

```
int isalpha (c)                 int isspace (c)
int c;                          int c;

int isupper (c)                 int ispunct (c)
int c;                          int c;

int islower (c)                 int isprint (c)
int c;                          int c;

int isdigit (c)                 int isgraph (c)
int c;                          int c;

int isxdigit (c)                int iscntrl (c)
int c;                          int c;

int isalnum (c)                 int isascii (c)
int c;                          int c;

int iswalpha (c)                int iswupper (c)
wchar_t c;                      wchar_t c;

int iswlower (c)                int iswdigit (c)
wchar_t c;                      wchar_t c;

int iswxdigit (c)               int iswalnum (c)
wchar_t c;                      wchar_t c;

int iswspace (c)                int iswpunct (c)
wchar_t c;                      wchar_t c;

int iswprint (c)                int iswgraph (c)
wchar_t c;                      wchar_t c;

int iswcntrl (c)                int iswascii (c)
wchar_t c;                      wchar_t c;
```

Description
The **ctype** macros classify character-coded integer values by table lookup. Each of these macros returns a nonzero value for TRUE and 0 for FALSE.

The **ctype** character macros examine the least significant byte of the parameter passed to them. Their use should be limited to an ASCII environment.

The **isascii** macro is defined for all integer values. The other macros return a meaningful value only if **isascii** returns TRUE for the same **c**

value, or if **c** is EOF. (See " stdio" in topic 1.2.283 for information about the value EOF.)

Each of these macros also can be found in **libc.a** as subroutines.

The following list shows the set of values for which each macro returns a nonzero (TRUE) value:

isalpha **c** is a letter.

isupper **c** is an uppercase letter.

islower **c** is a lowercase letter.

isdigit **c** is a digit in the range [**0-9**].

isxdigit **c** is a hexadecimal digit in the range [**0-9**], [**A-F**] or [**a-f**].

isalnum **c** is alphanumeric (a letter or a digit).

isspace **c** is a space, tab, carriage return, new-line, vertical tab, or form-feed character.

ispunct **c** is a punctuation character (neither a control character nor alphanumeric).

isprint **c** is a printing character, ASCII space (040 or 0x20) through **~** (0176 or 0x7E).

isgraph **c** is a printing character, like **isprint** but, unlike **isprint**, **isgraph** returns FALSE (0) for the space character.

iscntrl **c** is an ASCII DEL character (0177 or 0x7F) or an ordinary control character (less than 040 or 0x20).

isascii **c** is an ASCII character whose value is in the range 0-0177 (0-0x7F), inclusive.

The wide character **ctype** macros classify characters according to the rules of the coded character set defined by character type information in the program's locale (category **LC_TYPE**). You should also call these wide character macros if you are in the **C** locale.

The following list specifies the set of values for which each wide character **ctype** macro returns a nonzero (TRUE) value:

iswalpha (c) **c** is an upper or lower case process code character.

iswupper (c) **c** is an upper case process code character.

iswlower (c) **c** is a lower case process code character.

iswdigit (c) **c** is a process code digit in the range [0-9].

iswxdigit (c) **c** is a hexadecimal process code digit in the range [0-9], [A-F], or [a-f].

iswalnum (c) **c** is a process code alphanumeric.

iswspace (c) **c** is a process code space, tab, carriage return, new-line,

vertical tab, or form-feed character.

iswpunct (c) **c** is a process code punctuation (neither a control chracter nor alphanumeric).

iswprint (c) **c** is a printable process code character.

iswgraph (c) **c** is a printable process code character like **iswprint**, but **iswgraph** returns FALSE (0) for the space character.

iswcntrl (c) **c** is an ordinary process code control character.

iswascii (c) **c** is a process code ASCII character.

Related Information
In this book: "NCctype" in topic 1.2.183, "ascii" in topic 2.4.2, "setlocale" in topic 1.2.251, and "data stream" in topic 2.4.3.

"Introduction to International Character Support" in *Managing the AIX Operating System* and the **ctab** command in *AIX Operating System Commands Reference*.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.56 curses

Purpose
Controls cursor movement and windowing.

Library
Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
#include <term.h>
```

Description

Note: The **curses** package of subroutines is included here only for compatibility with existing programs. For information about the enhanced screen-handling subroutine library, see "extended curses library" in topic 1.2.74. The system calls **curses** and **extended curses** cannot both be used by the same program.

The **curses** subroutine package updates the screen with reasonable optimization. The **term.h** header file is only needed if **terminfo** level routines are needed (see "Terminfo Level Subroutines" in topic 1.2.56.2).

In order to initialize the routines, the routine **initscr** must be called before any of the other routines that deal with windows and screens are used. The routine **endwin** should be called before exiting. To get character-at-a-time input without echoing, call the **nonl**, **cbreak**, and **noecho** routines. Most interactive, screen-oriented programs require the character-at-a-time input without echoing.

The full **curses** interface permits manipulation of data structures called ***windows***, which can be thought of as two-dimensional arrays of characters representing all or part of a screen. Default windows called **stdscr** and **curscr** are supplied, and others can be created with the **newwin** routine. Windows are referred to by variables declared **WINDOW ***. The type **WINDOW** is defined in **curses.h** to be a C structure. These data structures are manipulated with the routines described below, among which the most basic are **move** and **addch**, which modify **stdscr.** After manipulating the data structures, the **refresh** subroutine is called, which updates the screen to look like **stdscr**. Routines beginning with the new definition of **w** allow window specification. Routines not beginning with a **w** affect **stdscr**. For further information about video mode support, see "Configuring the Virtual Terminal" in topic 2.5.11.8.

Minicurses is a subset of **curses** that does not allow manipulation of more than one window. To invoke this subset, use **-DMINICURSES** as a **cc** option. This level is smaller and faster than the full **curses**.

If the environment variable **TERMINFO** is defined, any program using **curses** checks for a local terminal definition before checking in **/usr/lib/terminfo**. For example, if **TERM** is set to **vt100**, the compiled file is normally found in **/usr/lib/terminfo/v/vt100**. (The directory name **v** is copied from the first letter of **vt100** to avoid creating huge directories.) If, for example, **TERMINFO** is set to **/usr/mark/myterms**, **curses** first checks **/usr/mark/myterms/v/vt100**. If this file does not exist, **curses** then checks **/usr/lib/terminfo/v/vt100**. This is useful for developing experimental definitions or when write permission in **/usr/lib/terminfo** is not available.

The following parameter names are of the type.

```
int win, p1, p2, p3;
char *str;
    int p1, p2, p3;
```

Note: The plotting library, **plot** and the curses library, **curses** both use the names **erase** and **move**. The **curses** versions are macros. If you need both libraries, put the **plot** code in a different source file than the **curses** code, or include the following statements in the **plot** code:

```
  #undef move()
  #undef erase()
```

Subtopics
1.2.56.1 Routines
1.2.56.2 Terminfo Level Subroutines
1.2.56.3 termcap Compatibility Routines
1.2.56.4 Attributes
1.2.56.5 Function Keys

1.2.56.1 Routines

The routines listed here can be called when using the full **curses**. Those marked with an asterisk can be called when using **minicurses**.

Note: In the following routines, **flag** is a Boolean variable and should have a value of TRUE or FALSE.

addch(int ch);* Add the character **ch** to **stdscr** (like **putchar**), wrapping to the next line at the end of a line.

waddch(window *win, int ch);
Add the character **ch** to **win**.

mvwaddch(window *w, int y, int x, int ch);
Move cursor position to (**y**, **x**) then add the character **ch** to **win**.

addstr(int str);* Call **addch** with each character in **str**.

mvaddstr(int y, int x; int str);
Move cursor position to (**y**, **x**) then add **str**.

waddstr(window *win, int str);
Add the string **str** to **win**.

mvwaddstr(window *win, int y, int x, int str);
Move cursor position to (**y**, **x**) then add the string **str** to **win**.

attroff(int attrs); Turn off the attributes named in **attrs**.

attron(int attrs); Turn on the attributes named in **attrs**.

attrset(int attrs); Set current attributes to those specified in **attrs**.

baudrate ( );* Set current terminal speed.

beep ( );* Sound beep on terminal.

box(window *win, int vert, int hor);
Draw a box around edges of **win**. The **vert** and **hor** parameters are the characters to use for vertical and horizontal edges of the box.

cbreak ( );* Set cbreak mode.

nocbreak ( );* Unset cbreak mode.

clear ( ); Clear **stdscr**.

clearok(window *win, int bf);
Clear screen before next redraw of **win**.

clrtobot ( ); Clear to bottom of **stdscr**.

clrtoeol ( ); Clear to end of line on **stdscr**.

delay_output(int ms);*
Insert **ms** millisecond pause in output.

nodelay(window *win, int bf);
Enable **nodelay** input mode through **getch**.

delch ( ); Delete a character from **stdscr**.

deleteln ( ); Delete a line from **stdscr**.

delwin(window *win); Delete window **win**.

doupdate ( ); Update screen from all **wnoutrefresh**.

echo ( );* Set echo mode.

noecho ( );* Unset echo mode.

endwin ( );* End window modes.

erase ( ); Erase **stdscr**.

erasechar ( ); Return user's erase character.

fixterm ( ); Restore terminal to **in curses** state.

flash ( ); Flash screen or beep.

flushinp ( );* Throw away any type-ahead.

getch ( );* Get a character from **tty**.

getstr(int *str); Get a string through **stdscr**.

gettmode ( ); Establish current **tty** modes.

getyx(window *win, int y, int x);
Get (**y**, **x**) coordinates.

has_ic ( ); Returns the value of TRUE if terminal can insert characters.

has_il ( ); Returns the value of TRUE if terminal can insert lines.

idlok(window *win, int bf);*
Use terminal's insert/delete line if **flag**!=0.

inch ( ); Get character at current (**y**, **x**) coordinates.

initscr ( );* Initialize screens.

insch(c); Insert a character.

insertln ( ); Insert a line.

intrflush(window *win, int bf);*
Interrupt flush output if **flag** is true.

keypad(WINDOW *win, int flag);*
Enable keypad and function keys.

killchar ( ); Return current user's **kill** character.

leaveok(window *win, int flag);
Permit cursor to be left anywhere after refresh if FALSE for **win**; otherwise cursor must be left at current position.

longname ( ); Return verbose name of terminal.

meta(window *win, int flag);
Allow metacharacters on input if FALSE.

move(int y, int x); Moves cursor to (**y**, **x**) on **stdscr**.

mvaddch(int y, int x, int ch);
Move cursor position to (**y**, **x**) then add **ch**.

mvcur(int oldrow, int oldcol, int newrow, int newcol);
Move cursor from current position to another position.

mvdelch(int y, int x);
Move cursor position to (**y**, **x**) then delete a character.

mvgetch(int y, int x);
Move cursor position to (**y**, **x**) then get a character from **tty**.

mvgetstr(int y, int x, int str);
Move cursor position to (**y**, **x**) then get a string through **stdscr**.

mvinch(int y, int x,);
Move cursor position to (**y**, **x**) then get the character at current (**y**, **x**) coordinates.

mvinsch(int y, int x, int c);
Move cursor position to (**y**, **x**) then insert the character **c**.

mvprintw(int y, int x, int fmt, int args);
Move cursor position to (**y**, **x**) then get **printf** on **stdscr**.

mvscanw(int y, int x, int fmt, int args);
Move cursor position to (**y**, **x**) then scan through **stdscr**.

mvwdelch(window, *win, int y, int x);
Move cursor position to (**y**, **x**) then delete a character from **win**.

mvwgetch(window *win, int y, int x);
Move cursor position to (**y**, **x**) then get a character through **win**.

mvwgetstr(window *win, int y, int x, int str);
Move cursor position to (**y**, **x**) then get a string through **win**.

mvwin(window *win, int by, int bx);
Move **win** so that the upper left-hand corner is located at (**y**, **x**).

mvwinch(window *win, int y, int x);
Move cursor position to (**y**, **x**) then get the character at current (**y**, **x**) in **win**.

mvwinsch(window *win, int y, int x, int c);
Move cursor position to (**y**, **x**) then insert the character **c** into **win**.

mvwprint(window *win, int y, int x, char *fmt, int args);
Move cursor position to (**y**, **x**) then **printf** on **stdscr**.

mvwscanw(window *win, int y, int x, char *fmt, int args);
Move cursor position to (**y**, **x**) then **scanf** through **stdscr**.

window * newpad(int nlines, int ncols);
Create a new pad with given dimensions.

struct screen *newterm(char *type, FILE *outfd, FILE *infd);
Set up new terminal of given type to output on **fd**.

window *newwin(int nlines, int ncols, int by, int bx);
Create a new window.

nl ( )*; Set newline mapping.

nonl ( )*; Unset newline mapping.

overlay(window *win1, window *win2);
Overlay **win1** on **win2**.

overwrite(window *win1, window *win2);
Overwrite **win1** on top of **win2**.

printw(char *fmt, va_dcl);
Printw on **stdscr**.

raw ( );* Set raw mode.

refresh ( );* Make **curscr** look like **stdscr**.

prefresh(window *pad, int pminrow, int pmincol, int sminrow, int smincol, int
smaxrow, int smaxcol);
Refresh from **pad** starting with given upper left corner of pad with output to given portion of screen.

pnoutrefresh(WINDOW *pad, int pminrow, int pmincol, int sminrow, int
smincol, int
smaxrow, int smaxcol);
Refresh like **prefresh**, but with no output until **doupdate** is called.

noraw ( );* Unset **raw** mode.

resetterm ( );* Set **tty** modes to **out of curses** state.

resetty ( );* Reset **tty** flags to stored value.

saveterm ( );* Save current modes as **in curses** state.

savetty ( );* Store current **tty** flags.

scanw(char *fmt, va_dcl);
Scanf through **stdscr**.

scroll(window *win); Scroll **win** one line.

scrollok(window *win, int bf);
Allow terminal to scroll if **flag**=FALSE.

set_term(char *type);
Enable talk to terminal **new**.

setscrreg(int t, int b)
Set user scrolling region to lines **short t** through **short b**.

setterm(char *type); Establish terminal with a given type.

standend ( )* Clear standout mode attribute.

standout ( )* Set standout mode attribute.

subwin(window *orig, int num_lines, int num_cols, int begy, int begx);
Create a subwindow.

touchwin(window *win);
Forces the next call to refresh( ) to write the entire window.

traceoff ( ) Turn off debugging trace output.

traceon ( ) Turn on debugging trace output.

typeahead(int fd); Check file descriptor **fd** to check type-ahead.

unctrl(ch)* Use printable version of **ch**.

wattroff(window *win, int attrs);
Turn off **attrs** in **win**.

wattron(WINDOW *win, int attrs);
Turn on **attrs** in **win**.

wattrset(window *win, int attrs);
Set attributes in **win** to **attrs**.

wclear(WINDOW *win); Clear **win**.

wclrtobot(WINDOW *win);
Clear to bottom of **win**.

wclrtoeol(WINDOW *win);
Clear to end of line on **win**.

wdelch(WINDOW *win); Delete the character at the current cursor coordinates in **win**.

wdeleteln(WINDOW *win);
Delete line from **win**

werase(WINDOW *win); Erase **win**.

wgetch(WINDOW *win); Get a character through **win**.

wgetstr(WINDOW *win, char *str);
Get the string **str** through **win**.

winch(WINDOW *win); Get the character at current cursor coordinates in **win**.

winsch(WINDOW *win, chtype c);
Insert the character **c** into **win**.

winsertln(WINDOW *win);
Insert line into **win**.

wmove(WINDOW *win, int y, int x);
Move the cursor to (**y**, **x**) coordinates on **win**.

wnoutrefresh(WINDOW *win);
Refresh but no screen output.

wprintw(WINDOW *win, char *fmt, va_dcl);
printf on **win**.

wrefresh(WINDOW *win);
Make screen look like **win**.

wscanw(WINDOW *win, char *fmt, va_dcl);
scanf through **win**.

wsetscrreg(WINDOW *win, int t, int b);
Set scrolling region of **win** to lines **short t** through **short b**.

wstandend(WINDOW *win);
Clear standout attribute in **win**.

wstandout(WINDOW *win);
Set standout attribute in **win**.

1.2.56.2 Terminfo Level Subroutines

These routines should be called by programs that have to deal directly with the **terminfo** data base. Due to the low level of this interface, its use is discouraged. The header files **curses.h** and **term.h** should be included (in that order) to get the definitions for these strings, numbers, and flags. You should call **setupterm** before using any of the other **terminfo** subroutines. This defines the set of terminal-dependent variables defined in the **terminfo** file.

If the program needs only one terminal, you can specify the **-DSINGLE** flag to the C compiler. This results in static references instead of dynamic references to capabilities. The result is smaller code, but only one terminal can be used at a time for the program.

Capabilities with a Boolean value have the value 1 if the capability is present and 0 if it is not. Numeric capabilities have a value of -1 if the capability is missing and a value of 0 or greater if it is present. String capabilities have a NULL value if the capability is missing and otherwise have type **char *** and point to a character string that contains the capability. Special character codes that use the backslash and circumflex characters (\ and ^) are transformed into the appropriate ASCII characters. Padding information of the form **$<time>**, and parameter information beginning with **%** (percent) are left uninterpreted. The **tputs** routine interprets padding information and **tparm** interprets parameter information.

All **terminfo** strings (including the output of **tparm**) should be printed with **tputs** or **putp**. Before exiting, **reset_shell_mode** should be called to restore the **tty** modes. Programs desiring shell escapes can call **reset_shell_mode** before the shell is called and **reset_prog_mode** after returning from the shell.

delay_output (int ms);
Sets the output delay, in milliseconds.

def_prog_mode ( );
Saves the current terminal mode as program mode, in **cur_term->Nttyb**.

def_shell_mode ( );
Saves the shell mode as normal mode, in **cur_term->Ottyb**.
def_shell_mode is called automatically by **setupterm**.

putp(char *str);
Calls **tputs(str, 1, _outchar)**.

reset_prog_mode ( );
Puts the terminal into program mode.

reset_shell_mode ( );
Puts the terminal into shell mode. All programs must call **reset_shell_mode** before they exit. The higher-level routine **endwin** automatically does this.

setupterm(char *term, int filenum, int *erret);
Reads in the data base. **term** is a character string that specifies the terminal name. If **term** is 0, then the value of the **TERM** environment variable is used.
One of the following status values is stored into the integer pointed to by **erret**:

1 Successful completion
0 No such terminal
-1 An error occurred while locating the **terminfo** data base.

If the **erret** parameter is 0, then no status value is returned, and an error causes **setupterm** to print an error message and exit, rather than return. **filenum** is the file descriptor of the terminal being used for output. **setupterm** calls **termdef** to determine the number of lines and columns on the display. If **termdef** cannot supply this information, then **setupterm** uses the values in the **terminfo** data base. The simplest call is **setupterm(0, 1, 0)**, which uses all the defaults.

After the call to **setupterm**, the global variable **cur_term** is set to point to the current structure of terminal capabilities. It is possible for a program to use more than one terminal at a time by calling **setupterm** for each terminal and saving and restoring **cur_term**.

The **setupterm** subroutine also initializes the global variable **ttytype**, an array of characters to the value of the list of names for the terminal. The list comes from the beginning of the **terminfo** description.

char *tparm(char *str, int p1, int p2, ... int p9)
Instantiates the string **str** with parameters **p[i]**. The character string returned has the given parameters applied.

tputs(char *cp, int affcnt, int (*outc) ( ));
Applies padding information to string **cp**. **affcnt** is the number of lines affected, or 1 if not applicable. **outc** is a **putchar**-like routine to which the characters are passed one at a time.

Some strings are of a form like **$<20>**, which is an instruction to pad for 20 milliseconds.

vidputs(int newmode, int (*outc) ( ));
Outputs the string to put terminal in video attribute mode **attrs**. Characters are passed to the **putchar**-like routine **outc**. The **attrs** are defined in **<curses.h>**. The previous mode is retained by this routine.

vidattr(int newmode);
Like **vidputs**, but outputs through **putchar**.

1.2.56.3 termcap Compatibility Routines

These routines are included for compatibility with programs that require **termcap**. Their parameters are the same as for **termcap**, and they are emulated using the **terminfo** data base.

int tgetent(char *bp, char *name);
Looks up the **termcap** entry for **name**. **name** is a terminal name; **bp** is ignored. Calls **setupterm**.

int tgetflag(char *id);
Returns the Boolean entry for **id**. **id** is a 2-character string that contains a **termcap** identifier.

int tgetnum(char *id);
Returns the numeric entry for **id**. **id** is a 2-character string that contains a **termcap** identifier.

char * tgetstr(char *id, char *area);
Returns the string entry for **id**. **id** is a 2-character string that contains a **termcap** identifier. The **area** parameter is ignored.

char * tgoto(char *cap, int col, int row);
Applies parameters to the given **cap**. Calls **tparm**.

tputs(char *cp, int affcnt, int (*outc) ( ));
Applies padding to **cap** calling **outc** as **putchar**.

1.2.56.4 Attributes

The following video attributes can be passed to the routines **attron**, **attroff**, and **attrset**. Not all attributes are currently supported on all terminals (see "Possible graphic renditions of VGA adapter" in topic 2.5.11.7.3).

A_STANDOUT	The terminal's best highlighting mode
A_UNDERLINE	Underlined
A_REVERSE	Reverse video
A_BLINK	Blinking
A_DIM	Half bright
A_BOLD	Extra bright or bold
A_INVIS	Invisible (blanked or zero-intensity)
A_PROTECT	Protected
A_ALTCHARSET	Alternate character set
A_NORMAL	Normal attributes

1.2.56.5 Function Keys

The following function keys might be returned by **getch** if **keypad** has been enabled. Note that not all of these are currently supported due to lack of definitions in **terminfo**, or due to the terminal not transmitting a unique code when the key is pressed.

KEY_BREAK	Break key (unreliable)
KEY_DOWN	Down-arrow key
KEY_UP	Up-arrow key
KEY_LEFT	Left-arrow key
KEY_RIGHT	Right-arrow key
KEY_HOME	Home key
KEY_BACKSPACE	Backspace (unreliable)
KEY_F(n)	Function key F**n**, where **n** is an integer from 0 to 12
KEY_DL	Delete line
KEY_IL	Insert line
KEY_DC	Delete character
KEY_IC	Insert character or enter insert mode
KEY_EIC	Exit insert character mode
KEY_CLEAR	Clear screen
KEY_EOS	Clear to end of screen
KEY_EOL	Clear to end of line
KEY_SF	Scroll 1 line forward
KEY_SR	Scroll 1 line backwards (reverse)
KEY_NPAGE	Next page
KEY_PPAGE	Previous page
KEY_STAB	Set tab
KEY_CTAB	Clear tab
KEY_CATAB	Clear all tabs
KEY_ENTER	Enter or send (unreliable)
KEY_SRESET	Soft (partial) reset (unreliable)
KEY_RESET	Reset or hard reset (unreliable)
KEY_PRINT	Print or copy
KEY_LL	Home down or bottom (lower left)
KEY_A1	Upper left key of keypad
KEY_A3	Upper right key of keypad
KEY_B2	Center key of keypad
KEY_C1	Lower left key of keypad
KEY_C3	Lower right key of keypad

Related Information
In this book: "extended curses library" in topic 1.2.74, "termdef" in topic 1.2.302, and "terminfo" in topic 2.3.59.

1.2.57 cuserid

Purpose
Gets the alphanumeric user name associated with the current process.

Library
Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>

char *cuserid (s)
char *s;
```

Description
The **cuserid** subroutine generates a character string representing the user name of the owner of the current process.

If the **s** parameter is a NULL pointer, then the character string is stored into an internal static area, the address of which is returned.

If the **s** parameter is not a NULL pointer, then the character string is stored into the array pointed to by the **s** parameter. This array must contain at least **L_cuserid** characters. **L_cuserid** is a constant defined in the **stdio.h** header file.

If the user name cannot be found, the **cuserid** subroutine returns a NULL pointer; if the **s** parameter is not a NULL pointer, then a null character (**'\0'**) is stored into **s**[0].

Related Information
In this book: "getlogin" in topic 1.2.103, "getpwent, getpwuid, getpwnam, setpwent, endpwent" in topic 1.2.114, and " stdio" in topic 1.2.283.

1.2.58 dbm

Purpose
Performs data base operations.

Library
Database Library (**libdbm.a**)

Syntax

```
int dbminit (file)
char *file;

datum fetch (key)
datum key;

int store (key, content)
datum key, content;

int delete (key)
datum key;

datum firstkey ( )

datum nextkey (key)
datum key;

typedef struct
{
   char  *dptr;
   int   dsize;
} datum;
```

Description
The **dbm** subroutines maintain a data base of **key-content** pairs. These subroutines can handle very large data bases and access keyed items in one or two file-system accesses.

The **key** parameter is a pointer to data specified by the **content** parameter. The sum of the sizes of the **key-content** pairs must not exceed the internal block size of 1024 bytes. All **key-content** pairs that hash together must fit on a single block. The **store** subroutine returns an error if a disk block fills with inseparable data.

The **key** and the **content** parameters are described by the **typedef datum** structure. The **datum** structure refers to a string of bytes, the length of which is specified by the **dsize** field. The string is pointed to by the **dptr** field. The **dptr** pointers returned by these subroutines point to static storage that changes with subsequent calls. The strings can contain binary data or normal ASCII characters.

The data base is stored in two files. One file is a directory that contains a bit map and is suffixed with **.dir**. The second file contains all data and is suffixed with **.pag**. The **.pag** file contains holes that increase its apparent size to about four times its actual size. You cannot copy a **.pag** file using the standard utilities such as **cp** and **cat** without first filling these holes.

Before you can access a data base, you must open the data base with the **dbminit** subroutine. The **file.dir**, and **file.pag** files must already exist before you call **dbminit**. You can create an empty data base by creating zero-length **.dir** and **.pag** files.

After the data base is opened with the **dbminit** subroutine, you can use the **fetch** subroutine to access the data that is pointed to by the **key** parameter. You can use the **store** subroutine to write the data specified by the **content** parameter to a file and to specify the key to be used to access that data with the **key** parameter.

The **delete** subroutine removes the key specified by the **key** parameter and the data to which that key points. The **delete** subroutine does not

actually reclaim the file space, but it does make it available for reuse.

The **firstkey** and **nextkey** subroutines make a linear pass through all of the keys in a data base. The **firstkey** subroutine returns the first key in the data base. The **nextkey** subroutine returns the next key in the data base. For example, the following code makes a linear pass through a data base:

```
for (key = firstkey(); key.dptr != NULL; key = nextkey(key))
   {
   ...
   }
```

The order of keys that are returned by **firstkey** and **nextkey** depend on the hashing function.

Return Value

All of the **dbm** subroutines that return an **int** value return 0 upon successful completion, and they return a negative value if an error occurs. Subroutines that return a **datum** value indicate an error by setting the **dptr** field to NULL.

1.2.59 difftime

Purpose
Computes time difference.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <time.h>

double difftime (time2,time1)
time_t time2, time1;
```

Description
The **difftime** macro computes the difference between **time2** and **time1**. The **difftime** macro returns the elapsed time in seconds from **time1** to **time2** as a double precision number. Type **time_t** is defined in the **time.h** header file.

Example
The following example shows a timing application using **difftime**. The example calculates how long it takes to find the prime numbers from 2 to 10000.

```
  #include <time.h>
  #include <stdio.h>
  #define runs 1000
  #define arr_size 10000

  int mark[arr_size];

  main  ()
  {
      time_t start, finish;
      int i, loop, n, num;

      time (&start);
      for  (loop = 0; loop < runs; loop++)
         for  (n = 0; n < arr_size; n++)
            mark [n] = 0;
         for  (num = 0, n = 2; n<arr_size; n++)
            if  (!mark[n]) {
               for  (i = 2*n; i <arr_size; i +=n)
                   mark[i] = -1;
               ++num;
               }
      time (&finish);
      printf ("\nProgram takes %f seconds to find %d primes.\n",
         difftime (finish,start)/runs,num);
  }
```

Subtopics
1.2.59.1 Output

1.2.59.1 Output

```
The program takes 0.106000 seconds to find 1229 primes.
```

1.2.60 directory: opendir, readdir, telldir, seekdir, rewinddir, closedir

Purpose
Performs operations on directories.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir (dirname)          void seekdir (dirp, loc)
char *dirname;                  DIR *dirp;
                                long loc;
struct dirent *readdir (dirp)
DIR *dirp;                      void rewinddir (dirp)
                                DIR *dirp;
long telldir (dirp)
DIR *dirp;                      int closedir (dirp)
                                DIR *dirp;
```

Description
The **opendir** subroutine opens the directory designated by the **dirname** parameter and associates a directory stream with it. The **closedir** subroutine terminates a directory stream and closes the underlying file descriptor.

The **opendir** subroutine returns a pointer to the **DIR** structure of the directory stream. NULL is returned when **dirname** cannot be accessed, or when not enough memory is available to hold the whole stream. If **dirname** is not a directory, NULL is returned and **errno** is set to ENOTDIR.

The **readdir** subroutine returns a pointer to the next directory entry. When it reaches the end of the directory, or when it detects an invalid **seekdir** operation, **readdir** returns NULL. The **telldir** subroutine returns the current location associated with the specified directory stream.

The **seekdir** subroutine sets the position of the next **readdir** operation on the directory stream.

Note: Values from **telldir** are valid only for the duration of the **opendir** operation from which the **DIR** pointer was derived. If a directory is closed and reopened, the position of the directory stream is reset. Therefore, if you want to continue reading from the location of the directory prior to close, you should save the value of a **telldir** made before closing the directory. After reopening the directory, use **seekdir** to determine the previous value of **telldir**.

The **rewinddir** subroutine resets the position of the specified directory stream to the beginning of the directory.

A -1 is returned if **dirp** does not refer to an open directory stream; otherwise, 0 is returned.

Warning: It is recommended that these subroutines be used to access a

directory rather than using the **open** and **read** system calls. See **ulimit** and **read** for a description of the special AIX behavior of the **read** system call on directories.

Examples

The following code illustrates a search of a directory for entry **name**:

```
  len = strlen(name);
  dirp = opendir(".");
  for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp))
          if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
                  closedir(dirp);
                  return FOUND;
          }
  closedir(dirp);
  return NOT_FOUND;
```

Error Conditions

The **opendir**, **readdir**, and **closedir** subroutines fail if one or more of the following are true:

EACCES Search permission is denied for any component of **dirname** or read permission is denied for **dirname**.

ENAMETOOLONG
The length of the **dirname** string exceeds **PATH_MAX**, or a pathname component is longer than **NAME_MAX**.

ENOENT The **dirname** argument points to the name of a file which does not exist.

ENOTDIR A component of **dirname** is not a directory.

EMFILE Too many file descriptors are currently open for the process.

ENFILE Too many files are currently open in the system.

ENOENT The **dirname** argument points to an empty string.

EBADF The **dirp** argument does not refer to an open directory stream.

Related Information

In this book: "close, closex" in topic 1.2.48, "lseek" in topic 1.2.161, "open, openx, creat" in topic 1.2.199, "read, readv, readx" in topic 1.2.224, "scandir" in topic 1.2.240, and "dir" in topic 2.3.16.

1.2.61 dirstat

Purpose
Gets file status information for multiple files in a directory.

Syntax

#include <sys/dirstat.h>

int dirstat (fildes, buf, bufsize)
int fildes;
struct dirstat *buf;
unsigned int bufsize;

Description
The **dirstat** system call returns status information on multiple files within a directory. Its arguments are a file descriptor to a directory which has been opened for reading, a pointer to a buffer, and the size of the buffer in bytes. The buffer should be large enough to hold at least one **dirstat** structure and one **NAME_MAX** + 1 (256) byte character string. On each call, **dirstat** packs as many **dirstat** structures as possible into the user's buffer, and advances the file pointer associated with **fildes**. It returns a count of the number of structures in the buffer, unless an error occurs, in which case a -1 is returned and the file pointer is not advanced.

When **dirstat** encounters a symbolic link, the link is not followed and status information about the link itself is returned. Similarly, **dirstat** does not slide through a hidden directory; status information about the hidden directory itself is returned. When **dirstat** encounters a mount point, status information is returned for the root directory of the mounted file system. In particular, when performing a **dirstat** call on the root directory of a mounted file system, both the "." and ".." entries correspond to the root directory of the mounted file system.

If **dirstat** encounters an error while attempting to get the file status of some particular inode within the directory, it marks the corresponding **dirstat** structure's **dir_stat.st_ss** field with a -1 and sets **dir_stat.st_errno** to the error number encountered. The only other fields valid when the **dir_stat.st_ss** field is -1 are **dir_nam_len, dir_rec_len, dir_stat.st_ino**, and the directory entry name returned by **DIRST_NAME**() macro.

The structure which **dirstat** returns is given below. The fields of the **stat** structure (**dir_stat**) are described in "stat.h" in topic 2.4.22. The **dir_nam_len** field gives the length of the name, including the trailing null character. In general, **dirstat** uses less than **NAME_MAX** characters for the name. Instead, **dirstat** puts the next structure on the first four-byte boundary following the null-terminated name field of the current structure. This makes it impossible to access the buffer as an array of **dirstat** structures. Programmers should use the **DIRST_ADV** macro provided in **dirstat.h** to advance a pointer through the list of structures. This macro uses the **dir_rec_len** field to allow for future expansion of the **dirstat** structure without requiring the recompilation of old programs.

Programmers should use the **DIRST_NAME** macro provided in **dirstat.h** to obtain the name associated with a particular entry in the list. The macro takes a pointer to an entry in the list, and returns a pointer to the associated name. This macro should always be used to access the name, since it uses the **dir_rec_len** field to allow for future expansion of the

dirstat structure without requiring the recompilation of old programs.

The **dirstat** structure is given below:

short dir_nam_len;	Length of name
short dir_rec_len;	Length of **dirstat** structure and name
struct stat dir_stat;	**Stat** structure similar to that returned by **statx**

Return Value
Upon successful completion, a count of **dirstat** structures in the buffer is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **dirstat** system call fails if one or more of the following are true:

EFAULT **buf** points outside the process's allocated address space.

EBADF **fildes** is not a valid open file descriptor.

ENOTDIR **fildes** is not a file descriptor for a directory open for read.

ESITEDN1 The site or sites on which the directory is stored are now down.

EINVAL **bufsize** is smaller than the smallest legal return value, that is, smaller than the size of **struct dirstat** plus the space required for the smallest legal file name.

ENAMETOOLONG
The directory contains a file whose name plus a **dirstat** structure does not fit in the provided **bufsize** bytes. This error cannot occur if **bufsize** is at least the size of **struct dirstat** + **NAME_MAX** + 1.

Errors returned in **dir_stat.st_errno** correspond to those returned by **statx**, see "statx, fstatx, stat, fstat, fullstat, ffullstat, lstat" in topic 1.2.282.

Related Information

In this book: "stat.h" in topic 2.4.22 and "statx, fstatx, stat, fstat, fullstat, ffullstat, lstat" in topic 1.2.282.

1.2.62 disclaim

Purpose
Disclaims content of a memory address range.

Syntax

```
#include <sys/shm.h>

int disclaim (addr, length, flag)
char *addr;
unsigned int length, flag;
```

Description

The **disclaim** system call marks an area of memory that has content that is no longer needed. This allows the system to discontinue paging the memory area.

The **addr** parameter points to the beginning of the memory area, and the **length** parameter specifies its length in bytes. The **flag** parameter must be the value **ZERO_MEM**, which indicates that each memory location in the address range is to be set to 0.

Return Value
Upon successful completion, the **disclaim** system call returns a value of 0. If it fails, it returns a value of -1 and sets **errno** to indicate the error.

Error Conditions

The **disclaim** system call fails if one or more of the following is true:

EFAULT The calling process does not have write access to the area of memory that begins at **address** and extends for **length** bytes.

EINVAL The value of the **flag** parameter is not valid.

Related Information
In this book: "shmat" in topic 1.2.258 and "shmctl" in topic 1.2.259.

1.2.63 drand48

Purpose
Generates uniformly distributed pseudo-random number sequences.

Library
Standard C Library (**libc.a**)

Syntax

```
double drand48 ( )

double erand48 (xsubi)
unsigned short xsubi[3];

long lrand48 ( )

long nrand48 (xsubi)
unsigned short xsubi[3];

long mrand48 ( );

long jrand48 (xsubi)
unsigned short xsubi[3];

void srand48 (seedval)
long seedval;

unsigned short *seed48 (seed16v)
unsigned short seed16v[3];

void lcong48 (param)
unsigned short param[7];
```

Description
This family of subroutines generates pseudo-random numbers using the linear congruential algorithm and 48-bit integer arithmetic.

The **drand48** and **erand48** subroutines return nonnegative double-precision floating-point values uniformly distributed over the range of **y** values such that 0.0 = **y** < 1.0.

The **lrand48** and **nrand48** subroutines return nonnegative long integers uniformly distributed over the range of **y** values such that 0 = **y** < 2(31).

The **mrand48** and **jrand48** subroutines return signed long integers uniformly distributed over the range of **y** values such that -2(31) = **y** < 2(31).

The **srand48**, **seed48** and **lcong48** subroutines initialize the random-number generator. Programs should invoke one of them before calling **drand48**, **lrand48** or **mrand48**. (Although it is not recommended practice, constant default initializer values are supplied automatically if the **drand48**, **lrand48** or **mrand48** subroutines are called without first calling an initialization subroutine.) The **erand48**, **nrand48** and **jrand48** subroutines do not require that an initialization subroutine to be called first.

All the subroutines work by generating a sequence of 48-bit integer values, **X[i]**, according to the linear congruential formula:

X sub <n + 1> = (aX sub n + c) sub <mod m> %%%% n ge 0

The parameter **m** = 2(48); hence 48-bit integer arithmetic is performed. Unless the **lcong48** subroutine has been called, the multiplier value **a** and the addend value **c** are:

a = '5DEECE66D' sub 16 = '273673163155' sub 8

c = 'B' sub 16 = '13' sub 8

The value returned by the **drand48**, **erand48**, **lrand48**, **nrand48**, **mrand48**, and **jrand48** subroutines is computed by first generating the next 48-bit **X[i]**

in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (most significant) bits of **X[i]** and transformed into the returned value.

The **drand48**, **lrand48** and **mrand48** subroutines store the last 48-bit **X[i]** generated into an internal buffer; that is why they must be initialized prior to being invoked.

The **erand48**, **nrand48** and **jrand48** subroutines require the calling program to provide storage for the successive **X[i]** values in the array pointed to by the **xsubi** parameter. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of **X[i]** into the array and pass it as a parameter.

By using different parameters, the **erand48**, **nrand48**, and **jrand48** subroutines allow separate modules of a large program to generate several **independent** sequences of pseudo-random numbers. In other words, the sequence of numbers that one module generates does **not** depend upon how many times the subroutines are called by other modules.

The initializer subroutine **srand48** sets the high-order 32 bits of **X[i]** to the 32 bits contained in its parameter. The low order 16 bits of **X[i]** are set to the arbitrary value 330E[16].

The initializer subroutine **seed48** sets the value of **X[i]** to the 48-bit value specified in the array pointed to by the **seed16v** parameter. In addition, **seed48** returns a pointer to a 48-bit internal buffer that contains the previous value of **X[i]**. that is used only by **seed48**. The returned pointer allows you to restart the pseudo-random sequence at a given point. Use the pointer to copy the previous **X[i]** value into a temporary array. Later you can call **seed48** with a pointer to this array to resume where the original sequence left off.

The **lcong48** subroutine specifies the initial **X[i]** value, the multiplier value **a**, and the addend value **c**. The parameter array elements **param**[0-2] specify **X[i]**, **param**[3-5] specify the multiplier **a**, and **param**[6] specifies the 16-bit addend **c**. After **lcong48** has been called, a subsequent call to either **srand48** or **seed48** restores the standard **a** and **c** as specified previously.

Related Information
In this book: "rand, srand" in topic 1.2.221.

1.2.64 dup

Purpose
Duplicates an open file descriptor.

Syntax

int dup (fildes)
int fildes;

Description
The **dup** system call returns a new file descriptor for the file descriptor pointed to by the **fildes** parameter. The **fildes** parameter is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call, or from the **socket** or **socketpair** subroutine. The **dup** system call returns a new file descriptor having the following in common with the original:

The same open file or pip
The same file pointer (that is, both file descriptors share one fil pointer)
The same access mode (read, write or read/write
The same file status flag
The same locks

The new file descriptor is set to remain open across **exec** system calls. If the Transparent Computing Facility is installed, the new file descriptor is also set to remain open across **rexec** and **run** system calls. (For more information about file control, see "fcntl, flock, lockf" in topic 1.2.78.)

The file descriptor returned is the lowest one available.

Return Value
Upon successful completion, a file descriptor (nonnegative integer) is returned. If the **dup** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **dup** system call fails if one or more of the following are true:

EBADF **fildes** is not a valid open file descriptor.

EMFILE Two hundred (200) file descriptors are currently open.

Related Information
In this book: "close, closex" in topic 1.2.48, "dup2" in topic 1.2.65, "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, "fcntl, flock, lockf" in topic 1.2.78, "open, openx, creat" in topic 1.2.199, "pipe" in topic 1.2.204, "socket" in topic 1.2.275, and "socketpair" in topic 1.2.276.

1.2.65 dup2

Purpose
Duplicates an open file descriptor.

Syntax

int dup2 (oldfd, newfd)
int oldfd, newfd;

Description
The **dup2** system call duplicates the file descriptor, **oldfd**, to the new file descriptor, **newfd**. When you specify a value of **newfd** that is already in use, the descriptor is deallocated as if a **close** system call had been issued.

The **oldfd** parameter is a small nonnegative integer index in the descriptor table. Its value must be less than the size of the descriptor table obtained by the **getdtablesize** subroutine.

Note: Since **newfd** and **oldfd** are duplicate references to an open file once the **dup2** system call has been called, the **read**, **write**, and **lseek** system calls move a single pointer into that file. Append mode and both non-blocking I/O and asynchronous I/O options are then shared between the references. Therefore, if you want to place a separate pointer in the file, you should issue an additional **open** system call to obtain a different object reference instead of using the **dup2** system call.

The new file descriptor is set to remain open across **exec** system calls. If the Transparent Computing Facility is installed, the new file descriptor is also set to remain open across **rexec** and **run** system calls. (For more information about file control, see "fcntl, flock, lockf" in topic 1.2.78.)

Return Value
When the call succeeds, a file descriptor (nonnegative integer) is returned. If the **dup2** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **dup2** system call fails if one or more of the following is true:

EBADF The **oldfd** parameter is not a valid open file descriptor.

EBADF The **newfd** parameter is not between 0 and 199, which is the valid range for file descriptors.

Related Information
In this book: "accept" in topic 1.2.9, "close, closex" in topic 1.2.48, "dup" in topic 1.2.64, "fcntl, flock, lockf" in topic 1.2.78, "getdtablesize" in topic 1.2.93, "open, openx, creat" in topic 1.2.199, "pipe" in topic 1.2.204, "socket" in topic 1.2.275, and "socketpair" in topic 1.2.276.

1.2.66 dustat

Purpose
Gets file system statistics.

Syntax

```
#include <sys/types.h>
#include <dustat.h>

int dustat (gfs, packno, buffer, length)
gfs_t gfs;
pckno_t packno;
struct dustat *buffer;
int length;
```

Description
The **dustat** system call returns information about a mounted file system. The **gfs** argument is a global file system number identifying a device containing a mounted file system. If the file system is not replicated, the **packno** argument is ignored; otherwise, it specifies the particular pack for which information is desired. The **buffer** is a pointer to a **dustat** structure, **length** bytes long, with the following format:

```
  fstore_t        du_fstore;      /* copy's fstore flags */
  commitcnt_t     du_hwm;         /* high-water mark for commits */
  commitcnt_t     du_lwm;         /* low-water mark for commits */
  daddr_t         du_fsize;       /* size of entire volume */
  union du_mix {
        daddr_t   dum_tfree;      /* number of free blocks */
        dev_t     dum_majmin      /* major-minor #'s of the device */
  } DU_mix;

  ino_t           du_tinode;      /* # of free inodes */
  short           du_bsize;       /* size of blocks */
  pckno_t         du_pckno;       /* pack # of this copy of */
                                  /* a replicated gfs */
  ino_t           du_isize;       /* address of first data block. */
  siteno_t        du_site;        /* site where this copy */
                                  /* is mounted. */
  unsigned short  du_flags;       /* from s_flags */


  char            du_inopb;       /* inodes per block */
  char            du_version;     /* data format of fs */
  char            du_packcnt;     /* number of valid entries in */
                                  /* du_dpacklst */
  char            du_dummy[5];    /* for future use */
  char            du_fsmnt[32];   /* name of this file system */
  char            du_fpack[8];    /* name of this physical vol */
  struct dpacklst du_dpacklst[MAXPACKNO];
                                  /* array of packlists, info on other */
                                  /* mounted copies of this gfs */


  The following defines are automatically made for easy access:

  #define  du_majmin      DU_mix.dum_majmin
  #define  du_tfree       DU_mix.dum_tfree
```

and the dpacklst structure consists of:

```
fstore_t        dpk_fstore;
siteno_t        dpk_site;
unsigned short  dpk_flags;
pckno_t         dpk_pack;
short           dpk_dummy;          /* future expansion */
```

The **dustat** system call interprets a pack number (**packno** argument) of 0 as the CSS's pack and a pack number of -1 as the local site's pack. In the latter case, the major-minor field (**du_majmin**) of the **dustat** structure is filled in; otherwise, the free block field (**du_tfree**) is filled in.

The **dustat** system call does not report the actual number of inodes allocated when run on nonprimary packs of replicated file systems. Only the primary copy will show the correct number of inodes allocated.

Return Value
Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **dustat** system call fails if one or more of the following are true:

ENOSTORE **gfs** specifies a replicated file system, and **packno** is -1, but there is no local copy of the file system.

EINVAL **gfs** is not in the range of valid file system numbers.

EINVAL The device is not mounted, or the specified pack is not available.

EFAULT **buffer** points outside the process's allocated address space.

EFAULT The **length** parameter to **dustat** is not the same as size of (**struct dustat**).

EINTR The call is interrupted by a signal.

ESITEDN1 The site which had this pack mounted has left the current partition.

ESITEDN2 The operation was terminated because a site failed.

Related Information
In this book: "ustat" in topic 1.2.320 and "fs" in topic 2.3.20.

1.2.67 ecvt, fcvt, gcvt

Purpose
Converts a floating-point number to a string.

Library
Standard C Library (**libc.a**)

Syntax

```
char *ecvt (value, ndigit, dchar,*gcvt)(value, ndigit, buf)
double value;               double value;
int ndigit, *decpt, *sign;  int ndigit;
                            char *buf;
char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;
```

Description
The **ecvt**, **fcvt**, and **gcvt** subroutines convert floating-point numbers to strings.

The **ecvt** subroutine converts the **value** parameter to a null-terminated string and returns a pointer to it. The **ndigit** parameter specifies the number of digits in the string. The low-order digit is rounded. **ecvt** sets the **int** pointed to by the **decpt** parameter to the position of the decimal point relative to the beginning of the string. (A negative number means the decimal point is to the left of the digits given in the string). The decimal point itself is not included in the string. The **ecvt** subroutine also sets the **int** pointed to by the **sign** parameter to a nonzero value if the **value** parameter is negative and sets it to 0 otherwise.

The **fcvt** subroutine functions identically to **ecvt**, except that **ndigit** produces the number of digits to the right of the decimal point only; if the number is =17, there may be truncation or imprecision.

The **gcvt** subroutine converts the **value** parameter to a null-terminated string, stores it in the array pointed to by the **buf** parameter, and then returns **buf**. **gcvt** attempts to produce a string of **ndigit** significant digits in FORTRAN F-format. If this is not possible, then E-format is used. **gcvt** suppresses trailing zeros. The string is ready for printing, complete with minus sign, decimal point, or exponent, as appropriate.

The **ecvt**, **fcvt**, and **gcvt** subroutines represent the following special values that are specified in ANSI/IEEE standard 754-1985 for binary floating-point arithmetic:

Quiet NaN	**QNaN**
Signalling NaN	**SNaN**
±&infinity.	**INF**

The sign associated with each of these values is stored into the **sign** parameter; zero can also be positive or negative.

Note: In the F-format **ndigit** is the number of digits desired after the decimal point. Very large numbers will produce a very long string of digits before the decimal point and then **ndigit** digits after the decimal point. Generally it is better to use **gcvt** or **ecvt** for large numbers.

Warning: All three subroutines store the strings in a static area of memory whose contents are overwritten each time one of the subroutines is called.

Related Information

In this book: "a64l, l64a" in topic 1.2.6, "frexp, ldexp, modf" in topic 1.2.85, "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf" in topic 1.2.208, and "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf, wsscanf" in topic 1.2.241.

1.2.68 end, etext, edata

Purpose
Defines the last location of a program.

Library
None

Syntax

```
extern end;
extern etext;
extern edata;
```

Description
The external names **end**, **etext**, and **edata** are defined by the loader for all programs. They are not subroutines, but identifiers associated with the following addresses:

etext	The first address following the program text
edata	The first address following the initialized data region
end	The first address following the data region that is not initialized.

The ***break value*** of the program is the first location beyond the data. When a program begins running, this location coincides with **end**. However, many factors can change the break value, including:

- The **brk** system call
- The **malloc** subroutine
- The standard input/output subroutine
- The **-p** flag on the **cc** command.

Therefore, use **sbrk(0)**, not **end**, to determine the break value of the program.

Related Information
In this book: "brk, sbrk" in topic 1.2.21, "malloc, free, realloc, calloc, valloc, alloca, mallopt, mallinfo" in topic 1.2.162, and " stdio" in topic 1.2.283.

The **cc** command in *AIX Operating System Commands Reference*.

1.2.69 erf, erfc

Purpose
Computes the error and complementary error functions.

Library
Math Library (**libm.a**)

Syntax

```
#include <math.h>

double erf (x)                  double erfc (x)
double x;                       double x;
```

Description
The **erf** subroutine returns the error function of **x**, defined as:

The **erfc** subroutine returns 1.0 - **erf(x**). The **erfc** subroutine is provided because of the extreme loss of relative accuracy if **erf(x**) is called for large values of **x** and the result is subtracted from 1.0. For example, 12 decimal places are lost when calculating (1.0 - **erf**(5)).

Error Conditions
The **erf** and **erfc** subroutines fail if the following is true:

EDOM The value of **x** is NaN.

Related Information
In this book: "cbrt, exp, expm1, log, log10, log1p, pow, sqrt" in topic 1.2.28.

1.2.70 errunix

Purpose
Logs application errors.

Library
Run-time Services Library (**librts.a**)

Syntax

```
int errunix (buf, cnt)
char *buf;
unsigned short cnt;
```

Description
The **errunix** subroutine invokes the application error device driver to record an error log entry. **errunix** is a C run-time subroutine. Device drivers should use the **errsave** kernel subroutine to log error messages.

If the error device driver is not open, **errunix** opens it. Then the error log entry is written to it.

The **buf** parameter points to a buffer that contains the following information:

1. A word (**int**) that contains the **class**, **subclass**, **mask**, and **type** of the message, as defined in the discussion of "error" in topic 2.5.7.

2. An **int** that specifies the number of words of dependent data for the error log entry, including this **int** itself.

3. Words that contain the dependent information for the error log entry. The number of dependent data words must be one less than the word count specified immediately before them.

The other fields of the error log header (length, date and time, time extended, and node name) are supplied for you automatically.

The **cnt** parameter specifies the number of bytes in the buffer pointed to by **buf**. The **cnt** parameter must be a multiple of 4.

Return Value
Upon successful completion, a value of 0 is returned. If the **errunix** subroutine fails, an error message is written to the standard error output, and a value of -1 is returned.

File

/dev/error

Related Information
In this book: "error" in topic 2.5.7.

1.2.71 exec: execl, execv, execle, execve, execlp, execvp

Purpose
Executes a file.

Syntax

```
int execl (path, arg0 [, argint.execv)(path, argv)
char *path, *arg0, *arg1,...char *path, *argv [ ];

int execle (path, arg0 [, arint.execve,(path, argv, envp)
char *path, *arg0, *arg1,...charn*path,;*argv [ ], *envp [ ];

int execlp (file, arg0 [, arint.execvp)(file, argv)
char *file, *arg0, *arg1,...char *file, *argv [ ];
```

Description
The **exec** system call, in all its forms, executes a new program in the calling process. This call does not create a new program, but overlays the current program with a new one, which is called the ***new process image***. The new process image file can be one of three file types:

An executable binary file in **a.out** format (see "a.out" in topic 2.3.2)

An executable text file that contains a shell procedure (only **execlp** and **execvp** allow this type of new process image file)

A file that names an executable binary file or shell procedure to b run.

The last of the types mentioned is recognized by a header with the syntax:

#! path [**string**]

The **#!** is the file's ***magic number***, which identifies the file type. The **path** parameter is the path name of the file to be executed. The **string** parameter is an optional character string that contains no tab or space characters. The header must be terminated with a new-line character. When invoked, the new process is passed **path** as **argv**[0]. This is followed by the optional parameter **string** and the name of the new process image file. The rest of the arguments passed are the same as those passed to the **exec** system call.

The parameters for the **exec** system calls are defined as follows:

path This parameter points to the path name of the new process image file.

file This parameter points to the name of the new process image file. Unless **file** is a full path name, the path prefix for the file is obtained by searching the directories named in the **PATH** environment variable. The initial environment is supplied by the shell.

Note that **execlp** and **execvp** take **file** parameters, but the rest of the **exec** system calls take **path** parameters. (For information about the environment, see "environment" in topic 2.4.6 and the **sh** and **csh** commands in *AIX Operating System Commands Reference*.)

arg0 [*, arg1,...*]
These parameters point to null-terminated character strings. The

strings constitute the argument list available to the new process. By convention, at least **arg0** must be present, and it must point to a string that is the same as **path** or its last component.

argv This parameter is an array of pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, **argv** must have at least one element, and it must point to a string that is the same as **path** or its last component. The last element of **argv** is a NULL pointer.

envp This parameter is an array of pointers to null-terminated character strings. These strings constitute the ***environment*** for the new process. The last element of **envp** is a NULL pointer.

When a C program is executed, it receives the following parameters:

```
main (argc, argv, envp)
int argc;
char *argv [ ], *envp [ ];
```

Here **argc** is the argument count, and **argv** is an array of character pointers to the arguments themselves. By convention, the value of **argc** is at least one, and **argv**[0] points to a string containing the name of the new process image file.

The **main** routine of a C language program automatically begins with a run-time start-off routine. This routine sets a global variable named **environ** so that it points to the environment array passed to the program in **envp**. You can access this global variable by including the following declaration in your program:

```
extern char **environ;
```

The **execl**, **execv**, **execlp**, and **execvp** system calls use **environ** to pass the calling process's current environment to the new process.

File descriptors open in the calling process remain open in the new process, except for those whose **close-on-exec** flag is set. For those file descriptors that remain open, the file pointer is unchanged. (For information about file control, see "fcntl, flock, lockf" in topic 1.2.78.)

If the new process requires shared libraries, **exec** attaches each shared library image to the new process address space. (See *AIX Programming Tools and Interfaces*.) Shared libraries are searched for in the directories listed in the **LIBPATH** environment variable.

The **exec** system calls reset all caught signals to the default action. Signals that cause the default action continue to do so after **exec**. Ignored signals remain ignored, the signal mask remains the same, and the signal stack state is reset. (For information about signals, see "sigaction, sigvec, signal" in topic 1.2.263.)

If the set-user-ID mode bit of the new process image file is set, **exec** sets the effective user ID of the new process to the owner ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process is set to the group ID of the new process image file. The real user ID and real group ID of the new process remain the same as those of the calling process. (For information about the set-ID modes, see "chmod, fchmod" in

topic 1.2.44.) The effective user ID and the effective group ID are saved (as the saved set-user-ID and the saved set-group-ID) for use by the **setuid** function and for signal delivery permissions.

The shared libraries attached to the calling process are not attached to the new process.

Profiling is disabled for the new process. (For information about profiling, see "profil" in topic 1.2.210.)

The new process inherits the following attributes from the calling process:

- Nice value (see "getpriority, setpriority, nice" in topic 1.2.111)
- Process I
- Parent process I
- Process group I
- **semadj** values (see "semop" in topic 1.2.245)
- TTY group ID (see "exit, _exit" in topic 1.2.73 and "sigaction, sigvec, signal" in topic 1.2.263)
- Trace flag (see request 0 of "ptrace" in topic 1.2.212)
- Time left until an alarm clock signal (see "alarm" in topic 1.2.14)
- Current director
- Root director
- **<LOCAL>** alias pathname (see "getlocal, setlocal" in topic 1.2.102)
- File mode creation mask (see "umask" in topic 1.2.314)
- File locks (see "fcntl, flock, lockf" in topic 1.2.78)
- System resource limits (see "getrlimit, setrlimit, vlimit" in topic 1.2.115 and "ulimit" in topic 1.2.313)
- **utime**, **stime**, **cutime**, and **cstime** (see "times" in topic 1.2.304)
- **xvers** string (see "getxvers, setxvers" in topic 1.2.129)
- Site path (see "getspath, setspath" in topic 1.2.122)
- Execution site permissions (see "getxperm, setxperm" in topic 1.2.128).

The name of the new process image file may refer to a hidden directory. Hidden directories are normally used in a Transparent Computing Facility cluster to enable execution of the correct process image for a given machine type. Without TCF, however, hidden directories can be used to allow execution of different versions of a program without changing the program name (see "getxvers, setxvers" in topic 1.2.129).

If the Transparent Computing Facility is installed, the following information also applies.

After the **exec** system call, the calling process executes on a site determined by the machine type on which the new process image must run and by the site path.

If the name of the new process image file refers to a hidden directory, **exec** first selects the new process image file by following the usual hidden directory path search rules, determined by prior calls to **getspath**. The selected new process image file is examined to determine on what type of site it may run. Then the site path is searched until the corresponding machine type or a site of the appropriate type is found. If a specific site is found, the **exec** system call executes a new program at that site. If an entry for the machine type is found, the **exec** system call executes a new program at a site of that type, using the local site if possible. The user must have permission to move processes to the destination site (see "getxperm, setxperm" in topic 1.2.128).

The user level code is responsible for maintaining a site path which is compatible with the user's permissions and the current network partition. If the contents of the site path cause the system to choose a new process image file for which there is no permissible site in the current partition, then the system call **exec** fails.

The **utime**, **stime**, **cutime**, and **cstime** (see "times" in topic 1.2.304) are preserved by the **exec** system call if the process remains on the same site. If it moves to a new site, they are changed accordingly to the following rules:

```
cutime += utime;

cstime += stime;

utime = 0;

stime = 0;
```

Note: Processes may not execute to another site if:

1. They have too many (85 or more) child processes.
2. They have a file open which is marked as being in error (for instance, the storage site is not on the cluster network)
3. They have made use of semaphores or messages operations (see "Semaphores, Message Queues, and Shared Memory Segments" in topic 1.2.2.10)
4. There are any TCP/IP sockets open (see "TCP/IP Communication" in topic 1.2.2.8).

Return Value
Upon successful completion, **exec** does not return because the calling process image is overlaid by the new process image. If **exec** returns to the calling process, then it returns the value -1 and sets **errno** to indicate the error.

Error Conditions
The **exec** system call fails and returns to the calling process if one or more of the following are true:

ENOENT One or more components of the new process image file's **path** name do not exist.

ENOTDIR A component of the **path** prefix of the new process image file is not a directory.

EACCES Search permission is denied for a directory listed in the **path** prefix of the new process image file.

EACCES The new process image file is not an ordinary file.

EACCES The mode of the new process image file denies execution permission.

ENOEXEC The **exec** system call is not an **execlp** or **execvp**, and the new process image file has the appropriate access permission but has an invalid magic number in its header.

ENOEXEC The new process image file has a valid magic number in its

header, but the header is damaged or is incorrect for the machine on which the file is to be run.

ETXTBSY The new process image file is a pure procedure (shared text) file that is currently open for writing by some process.

ENOMEM The new process requires more memory than is allowed by the system-imposed maximum **MAXMEM**.

E2BIG The number of bytes in the new process's argument list is greater than the system-imposed limit. This limit is defined as **NCARGS** in the **sys/param.h** header file.

EFAULT The **path**, **argv**, or **envp** parameter points to a location outside of the process's allocated address space.

In addition, some errors can occur when using the new process file after the old process image has been overwritten. These errors include problems in setting up new data and stack registers, problems in mapping a shared library, or problems in reading the new process file. Because returning to the calling process is not possible, the system sends the **SIGKILL** signal to the process when one of these errors occurs.

ESTALE The process's root or current directory is located in an NFS virtual file system that has been unmounted.

ENAMETOOLONG
A component of the **path** parameter exceeded **NAME_MAX** characters or the entire **path** parameter exceeded **PATH_MAX** characters.

ENOENT A hidden directory was named, but no component inside it matched the process's current site path list.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

ELOOP A loop of symbolic links was detected.

EIO A physical I/O error occurred.

If a shared library cannot be attached, one of more of the following is true:

ELIBMAX More than 10 shared libraries are specified by the new process image file.

ELIBSCN The specification of shared libraries in the **.lib** section of the new process image file is not in the correct format (see "a.out" in topic 2.3.2).

ELIBACC A shared library specified by the new process image file cannot be opened.

ELIBBAD A shared library file is not in correct **a.out** format (see "a.out" in topic 2.3.2).

If the Transparent Computing Facility is installed on your system, the **exec** system call can also fail if one or more of the following are true:

ESITEDN1 **path** cannot be accessed because a site went down.

ENOSTORE **path** is a name relative to the working directory, but no site which stores this directory is currently up.

EINTR A signal was caught during the system call.

ENOSTORE A component of **path** is replicated but is not stored on any site which is currently up.

ESITEDN1 The site chosen for the **exec** is down.

EPERM Execute permission is not granted for any of the sites chosen by the site path.

ELDWRG The **exec** system call tried to execute on a site that is the wrong machine type for the new process image.

ETABLE On either the new site or the old site, the system's PID-site table, which is used to keep track of remote processes and process groups, is full.

ELOCALONLY The calling process may not use **exec** to execute a new program at another site because it is using local only resources, such as semaphores, or it has too many child processes.

ENLDEV The process may not execute on the designated site because one of its open file descriptors is for a local-only object such as a socket or a non-**tty** character special file.

In addition, the process may be killed with a **SIGKILL** signal if a system error occurs very late in the process of reading in the new process image. These system errors include being out of text table space and getting a disk read error while reading the new process image file.

Examples

1. To run a command and pass it a parameter:

```
execlp("li", "li", "-al", 0);
```

   The **execlp** system call searches each of the directories listed in the **PATH** environment variable for the **li** command, and then it overlays the current process image with this command. **execlp** does not return, unless the **li** command cannot be executed. Note that this example does not run the shell command processor, so operations interpreted by the shell, such as using wildcard characters in file names, are not valid.

2. To run the shell to interpret a command:

```
execl("/bin/sh", "sh", "-c", "li -l *.c", 0);
```

   This runs the **sh** (shell) command with the **-c** parameter, which indicates that the following parameter is the command to be interpreted. (See the discussion of **sh** in *AIX Operating System Commands Reference* for details about this command.) This example uses **execl** instead of **execlp** because the full path name **/bin/sh** is specified, making a **PATH** search unnecessary.

   Running a shell command in a child process is generally more useful than simply using **exec**, as shown here. The simplest way to do this is

to use the **system** subroutine. See "system" in topic 1.2.298 for information about this subroutine.

3. The following is an example of a new process file that names a program to be run:

```
#! /bin/awk -f
{ for (i = NF; i > 0; --i)  print i }
```

If this file is named **reverse**, then typing the following command on the command line:

```
reverse chapter1 chapter2
```

causes the following command to be run:

```
/bin/awk -f reverse chapter1 chapter2
```

Note that the **exec** system calls use only the first line of the new process image file and ignore the rest of it. Also, **awk** interprets the text that follows a # (number sign) as a comment. (See the **awk** command in *AIX Operating System Commands Reference* for more information.)

Related Information

In this book: "alarm" in topic 1.2.14, "chmod, fchmod" in topic 1.2.44, "exit, _exit" in topic 1.2.73, "fcntl, flock, lockf" in topic 1.2.78, " fork, vfork" in topic 1.2.83, "getpriority, setpriority, nice" in topic 1.2.111, "profil" in topic 1.2.210, "ptrace" in topic 1.2.212, "semop" in topic 1.2.245, "shmat" in topic 1.2.258, "sigaction, sigvec, signal" in topic 1.2.263, "system" in topic 1.2.298, "times" in topic 1.2.304, "ulimit" in topic 1.2.313, "umask" in topic 1.2.314, "varargs" in topic 1.2.323, "a.out" in topic 2.3.2, and "environment" in topic 2.4.6.

The **csh** and **shlib2** commands in *AIX Operating System Commands Reference*.

1.2.72 exect

Purpose
Executes a file in trace mode.

Library
Berkeley Compatibility Library (**libbsd.a**)

Syntax

```
int exect (path, argv, envp)
char *path, argv [ ], *envp [ ];
```

Description
The **exect** subroutine is included for compatibility with older programs being traced with the **ptrace** command. Newer debuggers eliminate the requirement for this function. The program being executed is forced into hardware single-step mode.

Related Information
In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71 and "ptrace" in topic 1.2.212.

The **ptrace** command in *AIX Operating System Commands Reference*.

1.2.73 exit, _exit

Purpose
Terminates a process.

Syntax

void exit (status)	**void _exit (status)**
int status;	**int status;**

Description
The **exit** system call terminates the calling process and causes the following to occur:

All of the file descriptors open in the calling process are closed Since **exit** terminates the process, any errors encountered during these close operations go unreported.

If the parent process of the calling process is executing a **wait** system call, it is notified of the termination of the calling process and the low-order eight bits (that is, bits 0377 or 0xFF) of **status** are made available to it. See "wait, waitpid" in topic 1.2.325.

If the parent process of the calling process is not executing a **wait** system call, and if the parent hasn't set its **SIGCHLD** signal to **SIG_IGN**, then the calling process is transformed into a zombie process. A zombie process is a process that occupies a slot in the process table, but has no other space allocated to it either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information to be used by the **times** system call. (See "times" in topic 1.2.304 and the **sys/proc.h** header file.)

The parent process ID of all of the calling process's existing chil processes is set to -1. This is done to avoid confusion between processes which are the real child processes of the **init** process and processes which are orphaned; this is useful in a Transparent Computing Facility cluster. **getppid** will return a 1 if the parent ID is -1 (see "getpid, getpgrp, getppid" in topic 1.2.110). Zombie child processes of the exiting process are destroyed.

Each attached shared memory segment is detached and the value o **shm_nattach** in the data structure associated with its shared memory identifier is decremented by 1.

For each semaphore for which the calling process has set a **semadj** value, that **semadj** value is added to the **semval** of the specified semaphore. (See "semop" in topic 1.2.245 about semaphore operations.)

If the process has a process lock, text lock, or data lock, an **unlock** is performed. (See "plock" in topic 1.2.205.)

An accounting record is written on the accounting file if the system' accounting routine is enabled. (See "acct" in topic 1.2.11 for information about enabling accounting routines.)

If the calling process is a session leader, then the **SIGHUP** signal is sent to each process that has a process group ID equal to that of the calling process. In other words, if **exit** is called by the process group leader for the controlling terminal (typically the shell), then

SIGHUP is sent to all of the processes associated with that terminal.

Note: Note that since the C-shell starts each job in its own process group, jobs left in the background when a **login csh** exits, will not be sent **SIGHUP**.

If any child processes of the calling process are stopped, they ar sent **SIGHUP** and **SIGCONT** signals.

Locks set by the **lockf** system call are removed. (See "fcntl, flock, lockf" in topic 1.2.78 about file locks.)

The **exit** subroutine causes cleanup actions, including flushing of standard I/O buffers, to occur before the process exits. The **_exit** system call bypasses all cleanup.

Note: The effect of **exit** can be modified by the setting of the **SIGCHLD** signal in the parent process. See "sigaction, sigvec, signal" in topic 1.2.263.

Related Information
In this book: "acct" in topic 1.2.11, "sigaction, sigvec, signal" in topic 1.2.263, "times" in topic 1.2.304, and "wait, waitpid" in topic 1.2.325.

1.2.74 extended curses library

Purpose
Controls cursor movement and windowing.

Library
Extended Curses Library (**libcur.a**)

Syntax

#include <cur01.h>

Description

The Extended Curses subroutines control input and output to a workstation, performing optimized cursor movement, windowing, and other functions. This package is based on the **curses** subroutine package, which is included in most UNIX-compatible systems. The **curses** subroutines are also included in AIX for complete compatibility with existing programs (see "curses" in topic 1.2.56). However, **curses** and Extended Curses cannot both be used by the same program.

The enhancements provided by Extended Curses include:

- A wider range of display attribute
- Generalized drawing of boxe
- Terminal-independent input data processin
- Extended window contro
- Pane, panel, and field concept
- Support for extended character
- Handling of locator input

Subtopics

1.2.74.1 Terminology

window The internal representation of what a portion of the display may look like at some point in time. Windows can be any size from the entire display screen to a single character.

screen A window that is large as the display screen. A screen named **stdscr** is automatically provided.

terminal Sometimes called a terminal screen. A special screen that is the Extended Curses package's understanding of what the workstation's display screen currently looks like. The terminal screen is identified by a window named **curscr**, which should not be accessed directly by the user. Instead, changes should be made to **stdscr** (or a user-defined screen) and then **refresh** (or **wrefresh**) should be called to update the terminal.

presentation space The array that contains the data and attributes associated with a window.

pane An area of the display that shows all or part of the data contained in a presentation space associated with that pane.

active pane The pane in which the text cursor is positioned. A pane must be active before you can do input.

panel A group of one or more panes that are treated as a unit. The panes of a panel are displayed together, erased together, and usually represent a unit of information to a person using the application. A panel is represented on the display as a rectangular area that is tiled (completely filled) with panes.

field An area in a presentation space into which the program accepts input.

extended character A character other than 7-bit ASCII that can be represented in either 1 or 2 bytes. (See "data stream" in topic 2.4.3.)

NLSCHAR Represents an **mbchar_t** (unsigned long) character stored in four bytes. It is used for both interface data (arguments to library functions) and internal representation of a character.

When a double display width character needs to be displayed where only a single display width space is available, then a partial-character indicator is displayed instead. The partial-character indicator is the @ (at sign), and it is represented by the constant **pd_char**.

See the discussion of Extended Curses in *AIX Programming Tools and Interfaces*, and "Introduction to International Character Support" in *Managing the AIX Operating System* for more detailed information about these concepts.

1.2.74.2 Linking the Extended Curses Routines

The Extended Curses routines also call **terminfo** subroutines, which are located in the Curses Library (**libcurses.a**). Therefore, compile programs that use Extended Curses routines with the flags **-lcur** and **-lcurses**.

1.2.74.3 Header Files

The **cur00.h** header file replaces **curses.h** when converting programs that use the original **curses** package to Extended Curses.

All of the routines require the **cur01.h** header file.

The key codes returned by **getch** are defined in **cur02.h**.

The **cur03.h** header file defines attribute priority codes, and is not needed by application programs.

The **unctrl** routine requires **cur04.h**.

The routines that manage panes and panels (the routines whose name begin with **ec**) also require the **cur05.h** header file.

1.2.74.4 Naming Conventions

The new routines added to the original **curses** package begin with the letters **ec**.

Many routines operate on **stdscr**, the standard screen, by default. Corresponding routines that allow you to specify a window have the same name, prefixed with the letter **w**. For example, **addch** adds a character to **stdscr**, while **waddch** allows you to specify the window. Sometimes a routine beginning with **p** also exists, such as **paddch**, which allows you to specify a pane.

Some routines also allow you to specify cursor movement with the action to be performed. These routines have a prefix of **mv**. Thus, **addch** becomes **mvaddch**, **waddch** becomes **mvwaddch**, and **paddch** becomes **mvpaddch**. Each of these routines is equivalent to calling **move** or **wmove** before performing the operation.

The various prefixed forms of the routines are implemented as macros. In each case, the routine beginning with **w** is the base subroutine from which the others are defined.

1.2.74.5 Parameters

The following declarations serve for all of the routines:

```
    char ch *string;
    NLSCHAR xc;
    int line, col, firstline, firstcol;
    int numlines, numcols, numchars, length, mode;
    bool boolf;
    WINDOW *win, *win1, *win2, *oldwin, *newwin;
    PANE *pane;
    PANEL *panel;
```

1.2.74.6 Return Values

Unless otherwise noted, each routine returns a value of type **int** that is either OK (indicating successful completion) or ERR (if an error is encountered).

1.2.74.7 The Extended Curses Routines

The Extended Curses routines are listed here alphabetically, except that routines with **w**, **p**, and **mv** prefixes are listed with the corresponding routine that does not have these prefixes.

addch (*xc***)**
waddch (*win*, *xc***)**
paddch (*pane*, *xc***)**
mvaddch (*line*, *col*, *xc***)**
mvwaddch (*win*, *line*, *col*, *xc***)**
mvpaddch (*pane*, *line*, *col*, *xc***)**

The *xc* parameter is a value of type **NLSCHAR**.

The **addch** routine adds the **NLSCHAR** specified by the *xc* parameter on the window at the current (*line*, *col*) coordinates. **paddch** adds the character to the presentation space for the pane specified by the *pane* parameter. If the character is **'\n'** (new-line character), the line is cleared to the end, and the current (*line*, *col*) coordinates are changed to the beginning of the next line. A **'\r'** (return character) moves the current position to the beginning of the current line on the window. A **'\t'** (tab character) is expanded into spaces in the normal tabstop positions of every eighth column.

Adding a character to the lower right corner of a window that includes the lower right corner of the display causes many terminals to scroll the entire display image up one line. If adding a character or a character attribute causes such scrolling to occur, **addch** makes the change on the window but does not mark it for **wrefresh** purposes; **addch** returns the value ERR.

A double display width character must be added to **addch** in a single call. If adding a character would cause that character to split across two lines, the system appends a blank to the end of the current line and adds the entire character at the beginning of the following line. If an added character overwrites half an existing two-byte character, the system replaces the remaining half of that existing character with a blank.

addstr (*string***)**
waddstr (*win*, *string***)**
paddstr (*pane*, *string***)**
mvaddstr (*line*, *col*, *string***)**
mvwaddstr (*win*, *line*, *col*, *string***)**
mvpaddstr (*pane*, *line*, *col*, *string***)**

The **addstr** routine adds the string pointed to by the *string* parameter on the window at the current (*line*, *col*) coordinates. The string can contain single-shift control codes.

Upon successful completion, **addstr** returns OK and the current (*line*, *col*) coordinates point to the location just beyond the end of the string. The **addstr** routine returns ERR if an attempt is made to add a character to the lower right corner of a window that includes the lower right corner of the display. In this case, **addstr** writes as much of the string on the window as possible.

waddfld (*win*, *string*, *length*, *numlines*, *numcols*, *mode*, *xc***)**

The **waddfld** routine adds data to a field within a window. The current coordinates specify the upper-left corner of the field in the window. The *numlines* and *numcols* parameters specify the number of lines and columns in the field, respectively. The *length* parameter specifies the length of the data. The *mode* parameter specifies the attribute for the field output. The *xc* parameter specifies the **NLSCHAR** that is used to fill the remainder of the field after the data has been added to it. The **xc** parameter must be in single display width; otherwise ERR is returned.

If the string contains a **'\n'** (new-line character), the fill character is added to the remainder of the columns on that line of the field, and the remainder of the data is added starting at the first column of the next line of the field. A **'\r'** (return character) changes the current position to the beginning column of the field. A **'\t'** (tab character) is expanded with fill characters up to the next normal tabstop position within the field.

The **waddfld** routine follows the same rules as **addch** for adding double display width characters.

beep ( )

The **beep** routine sounds the speaker or bell at the workstation.

box (*win*, *System V*, *hor***)**
NLSCHAR *System V*, *hor*;

The **box** routine draws a box around the window specified by the *win* parameter. **box** uses the **NLSCHAR** specified by the *System V* parameter to draw the vertical sides of the box, and the **NLSCHAR** specified by the *hor* parameter for drawing the horizontal lines and corners.

If the window includes the lower right corner of the display and **scrollok** is not set, then the lower right corner of the box is not shown on the window, and the **box** routine returns ERR.

The **box** routine is a macro that invokes **superbox**.

cbox (*win***)**

The **cbox** routine draws a box around the window specified by the *win* parameter. The characters used are those defined in **/usr/lib/terminfo** (type 1 box characters) or defaulted during the initialization.

The **cbox** routine is implemented as a macro that invokes **superbox**.

The **cbox** routine returns ERR if the window includes the lower right corner of the display and **scrollok** is not set on.

chgat (*numchars*, *mode***)**
wchgat (*win*, *numchars*, *mode***)**

pchgat (*pane*, *numchars*, *mode***)**
mvchgat (*line*, *col*, *numchars*, *mode***)**
mvwchgat (*win*, *line*, *col*, *numchars*, *mode***)**
mvpchgat (*pane*, *line*, *col*, *numchars*, *mode***)**

The **chgat** routine changes the attributes of the next *numchars* characters on the window starting from the current (*line*, *col*) coordinates. The attributes are changed to the attributes specified by the *mode* parameter. This routine does not wrap around to the next line; however, specifying a value for the *numchars* parameter that would cause a line wrap is not an error.

The *mode* parameter is one or more of the attributes defined by the global attribute variables. More than one attribute may be specified by logically ORing them together. The following example changes the attributes of the next 10 characters to bold blue characters on a black background:

```
chgat (10, BOLD | F_BLUE | B_BLACK)
```

The range of columns to be changed should include entire characters. The **numchars** variable refers to the number of single or double width display characters. If the current (**line**, **col**) position is on the second column of a double display width character, clearing begins at position **col -1**.

The **chgat** routine returns ERR if the change forces scrolling and **scrollok** is not set on for the window.

clear ( )
wclear (*win***)**

The **clear** routine resets the entire **stdscr** window to blank characters. **clear** sets the current (*line*, *col*) coordinates to (0, 0).

clearok (scr, *boolf***)**
WINDOW *scr;

The **clearok** routine sets the **clear** flag for the screen specified by the **scr** parameter. If the *boolf* parameter is TRUE, the screen is cleared on the next call to **refresh** or **wrefresh**. If the *boolf* parameter is FALSE, the screen is not cleared on the next call to **refresh** or **wrefresh**. This only works on screens, and, unlike **clear**, does not alter the contents of the screen. If the **scr** parameter is **curscr**, the next **refresh** will cause a clear-screen, even if the window passed to **refresh** is not a screen.

The **clearok** routine returns ERR if the window is not a full-screen window.

clrtobot ( )
wclrtobot (*win***)**

The **clrtobot** routine erases the window from the current (*line*, *col*) coordinates to the bottom. **clrtobot** leaves the current (*line*, *col*) coordinates unchanged. This does not force a clear-screen sequence

on the next refresh. If the current (**line, col**) position is on the second column of a double display width character, clearing begins at position **col -1**.

The **clrtobot** routine always returns the value OK.

clrtoeol ( )
wclrtoeol (*win***)**

The **clrtoeol** routine clears the window from the current (*line*, *col*) coordinates to the end of the current line. The current (*line*, *col*) coordinates are not changed. If the current (**line, col**) position is on the second column of a double display width character, clearing begins at position **col -1**.

The **clrtoeol** routine always returns the value OK.

colorend ( )
wcolorend (*win***)**

The **colorend** routine returns the terminal to **NORMAL** mode. By default, **NORMAL** is usually defined as (**F_WHITE** | **B_BLACK**).

The **colorend** routine is a macro that invokes **xstandend**.

The **colorend** routine always returns the value OK.

colorout (*mode***)**
wcolorout (*win*, *mode***)**

The **colorout** routine sets the current standout bit-pattern of the window (*win***->_csbp**) to the attribute specified by the *mode* parameter. Characters added to the window after such a call will have *mode* as their attribute. The *mode* parameter is constructed by logically ORing together attributes that are declared in the **cur01.h** header file that are supported by the terminal.

The **colorout** routine overrides the current setting of the window and works in conjunction with almost all of the routines that cause output to be placed on the window.

The **colorout** routine is a macro that invokes **xstandout**.

The **colorout** routine always returns the value OK.

cresetty (*boolf***)**

The **cresetty** routine resets the terminal to the state saved by the last call to **csavetty**. Use this routine after the completion of a program that uses the terminal as a simple terminal. If the *boolf* parameter is TRUE, then the data in **curscr** is redisplayed.

crmode ( )
nocrmode ( )

The **crmode** routine turns off the ***canonical processing*** of input by the system device driver. When canonical processing is off, data is made available without waiting for a **'\n'** (new-line character). **nocrmode** enables canonical processing by the system device driver.

The **wgetch** routine, which is used for all Extended Curses input, forces the equivalent of **crmode** before requesting input if echoing is active, and reinstates the original status on exit. If you are using echo, you should issue a call to either **crmode** or **raw** to avoid multiple calls by **wgetch**.

The **crmode** routine differs from **raw** in that **crmode** has no effect on output data processing and does not disable signal processing by the device driver.

The **crmode** routine always returns the value OK.

csavetty (*boolf***)**

The **csavetty** routine saves the current Extended Curses state so that it can later be reset by **cresetty**. Use this routine before running a program that uses the terminal as a simple terminal. If the *boolf* parameter is TRUE, then the following status is set before saving the terminal status: **crmode**, **noecho**, **meta**, **nonl**, and **keypad** (TRUE).

delay

See **nodedelay** on page 1.2.74.7.

delch ( )
wdelch (*win***)**
mvdelch (*line*, *col***)**
mvwdelch (*win*, *line*, *col***)**

The **delch** routine deletes the character at the current (*line*, *col*) coordinates. Each character after the deleted character on the line shifts to the left, and the last characters become blank.

The **delch** routine always returns the value OK.

deleteln ( )
wdeleteln (*win***)**

The **deleteln** routine deletes the current line. Every line below the current line moves up, and the bottom line becomes blank. The current (*line*, *col*) coordinates remain unchanged.

The **deleteln** routine always returns the value OK.

delwin (*win***)**

The **delwin** routine deletes the window specified by the *win* parameter. All resources used by the deleted window are freed for future use.

If a window has a subwindow allocated inside of it, the deletion of the window does not affect the subwindow even though the subwindow is invalidated. Therefore, subwindows must be deleted before the outer windows are deleted.

The **delwin** routine always returns the value OK.

dounctrl (*boolf***)**

The **dounctrl** routine turns the printing of control characters on or off. If the *boolf* parameter is TRUE, then the printing is turned on; if FALSE, printing is turned off. By default, **dounctrl** processing is initially turned off. The **unctrl** routine defined in **cur04.h** is used to get the string of printable characters being printed. Control characters become the printable character represented by the control character plus 0x40, preceded by a ^ (circumflex).

drawbox (*win***, *line*, *col*, *numlines*, *numcols***)**

The **drawbox** routine draws a box with the upper left corner located at the position specified by the *line* and *col* parameters. The *numlines* parameter specifies the number of rows to be used by the box, and the *numcols* parameter specifies the number of columns to be used by the box.

The characters used to draw the box are either those specified in the **terminfo** file, or those defaulted at initialization.

The **drawbox** routine returns ERR if part or all of the box is outside the window, or the box addresses the lower right corner of the screen and **scrollok** is not on.

#include <cur05.h>
ecactp (*pane***, *boolf***)**

The **ecactp** routine specifies the active pane in a panel. The pane specified by the *pane* parameter is made the active pane if the *boolf* parameter is TRUE. If an active pane has been previously designated, then the border of that pane is reset to the inactive display mode, and the border of the pane specified by the *pane* parameter is set to the active display mode. If the *boolf* parameter is FALSE, then the border of the pane specified by the *pane* parameter is set to the inactive display mode.

#include <cur05.h>
ecadpn (*pane***, *win***)**

The **ecadpn** routine adds the window specified by the *win* parameter to the list of windows that can be presented in the pane specified by the *pane* parameter. No visible action occurs as a result of this routine. A call to **ecaspn** must be made after **ecadpn** to change the data associated with the pane display.

The **ecadpn** routine returns ERR if the system is unable to allocate the storage required.

#include <cur05.h>
ecaspn (*pane*, *win***)**

The **ecaspn** routine makes the window specified by the *win* parameter the current window for display in the pane specified by the *pane* parameter. A refresh call for the pane or panel is needed to cause the data to be presented on the display. The viewport associated with the pane is positioned with the top left corner of the viewport at the top left corner of the data for the window.

The **ecaspn** routine returns ERR if the window specified by the *win* parameter was not previously associated with this pane using **ecadpn**.

#include <cur05.h>
WINDOW *ecblks ( )

The **ecblks** routine returns a pointer to a window that is filled with blanks. This window is intended to be used as a filler for panes that have no real content. It requires less storage than normal windows because all lines will always contain blanks.

Do not modify or delete this window.

#include <cur05.h>
PANEL *ecbpls (*numlines*, *numcols*, *firstline*, *firstcol*, **title**, **divdim**, **border**,
short *numlines*, *numcols*, *firstline*, *firstcol*;
char *title;
char divdim, border;

The **ecbpls** routine builds a panel structure.

The *numlines* parameter specifies the panel size in rows.

The *numcols* parameter specifies the panel size in columns.

The *firstline* parameter specifies the panel's origin on the display's upper left corner row coordinate.

The *firstcol* parameter specifies the panel's origin on the display's upper left corner column coordinate.

The **title** parameter points to a title string. The title is shown centered in the top border. If no title is desired, this parameter should be NULL.

The **divdim** parameter specifies the dimension along which this panel is to be divided: either **Pdivtyv** (vertical) or **Pdivtyh** (horizontal).

The **border** parameter indicates whether or not this panel is to have a border: either **Pbordry** (yes) or **Pbordrn** (no).

The *pane* parameter points to the first pane that defines the divisions of this panel.

All parameters should be given as defined here. However, they are

not checked or used until a call is made to **ecdvpl**. An application may modify values put into this structure until it calls **ecdvpl**.

Upon successful completion, a pointer to the new panel is returned. **ecbpls** returns ERR if there is not enough storage available.

```
#include <cur05.h>
PANE *ecbpns (numlines, numcols, ln, ld, divdim, ds, du, border, lh, lv)
short numlines, numcols, ds;
PANE *ln, *ld, *lh, *lv;
char divdim, du, border;
```

The **ecbpns** routine builds a pane structure.

The *numlines* parameter specifies the number of rows in the presentation space for the pane.

The *numcols* parameter specifies the number of columns in the presentation space for the pane.

The **ln** parameter points to a neighboring pane either above or to the left.

The **ld** parameter points to the start of a chain for divisions of the pane.

The **divdim** parameter specifies the dimension of the pane along which division is to occur. This parameter is used if and only if the **ld** parameter is not NULL. Valid values for this parameter are **Pdivpnv** (vertical dimension) and **Pdivpnh** (horizontal dimension).

The **ds** and **du** parameters together specify the size of this pane as part of the division of a parent pane:

du	**Vertical or Horizontal Size of the Pane**
Pdivszc	The size is specified by the **ds** parameter.
Pdivszp	The size is **ds** ÷ 10000 of the available space. For example, if **ds** is 5000, then the row or column size is half of the available space.
Pdivszf	The pane has a floating size. The value of the **ds** parameter is not used.

If you specify NULL for the **ld** parameter or if you are not sure which value to use for **du**, specify **Pdivszf** for the **du** parameter.

The **border** parameter specifies whether or not this pane has a border: either **Pbordry** (yes) or **Pbordrn** (no).

The **lh** parameter points to a pane that is to scroll with this pane when the pane scrolls horizontally.

The **lv** parameter points to a pane that is to scroll with this pane when the pane scrolls vertically.

If the **ln** parameter is not NULL, the **divs** field of the pane structure being built receives the value that was in the **ln.divs** field. The **ln.divs** field is modified to point to the new pane structure being built.

If the **lh** and the **lv** parameters are not NULL, they are used to link the new structure to the specified structures and to link the specified structures to the new structure. The links thus created form a ring that includes all panes that scroll together.

Upon successful completion, a pointer to the new pane structure is returned. **ecbpns** returns ERR if a error is detected during processing.

#include <cur05.h>
ecdfpl (*panel*, *boolf***)**

The **ecdfpl** routine creates the Extended Curses **WINDOW** structures needed to define the specified panel.

At the time this routine is invoked, all size and location specifications of the panel and its constituent panes must be properly set. **ecdfpl** does not examine any of the division size specifications or the scroll link specifications.

The **fpane** pointer in the indicated **PANEL** structure must point to the first leaf pane for the panel, and the subsequent **nxtpn** pointers from that pane must form a loop back to the first leaf pane. (This is done by **ecdvpl**.)

A **WINDOW** structure is built for the panel specified by the *panel* parameter. This **WINDOW** has a size that corresponds to the size of the panel. For each of the panes in the subsequent chain, a separate **WINDOW** structure is built with a size that corresponds to the specified presentation space size or the viewport size, whichever is larger.

If borders are specified for any of the panes, those borders are drawn on the **WINDOW** for the panel. All corners are checked and, if needed, proper junction characters are used to draw the corner.

The *boolf* parameter indicates whether to suppress the creation of presentation spaces for the panes. If the value is TRUE, presentation spaces are not created. If FALSE, presentation spaces are created.

The **ecdfpl** routine returns ERR if sufficient storage is not available for the **WINDOW** structures being created.

#include <cur05.h>
ecdppn (*pane*, *oldwin*, *newwin***)**

The **ecdppn** routine adds, drops or replaces a presentation space for a pane.

First, if the *oldwin* parameter is not NULL, then **ecdppn** drops *oldwin* from the list of windows that are alternatives for the pane specified by the *pane* parameter. The previous association should have been established using **edadpn**. If the *oldwin* parameter is NULL, then no window is dropped.

Next, if the *newwin* parameter is not NULL, then **ecdppn** adds *newwin*

as a valid pane for this window, replacing *oldwin*, if it was associated with the pane specified by the *pane* parameter. (See **ecadpn** for a better way to add a pane).

The **ecdppn** routine always returns the value OK.

#include <cur05.h>
ecdspl (*panel***)**

The **ecdspl** routine releases all of the data structures associated with the panel specified by the *panel* parameter. The released data structures are returned to the free pool. The released data structures include the panel structure, all associated pane structures, any window structures associated with the panes, any auxiliary window structures associated with the panes, and all private control structures used by Extended Curses.

#include <cur05.h>
ecdvpl (*panel***)**

The **ecdvpl** routine assigns a real size and relative position to all the panes defined for the panel specified by the *panel* parameter. All of the panes must be linked to the panel. The structure of a tree is followed to determine the sizes for each pane.

The direction of the first set of divisions and the size of the first set of divisions is determined. This information is used to control the division algorithm. Using the size along the direction of division, first, the total space for the interior of panes is determined by counting the panes and their borders. Next, any panes with **fixed** size are given the space indicated by the **divsz** field in the pane structure. The remaining available space is then assigned to the panes that have specified a **proportional** size. Finally, any space that remains is assigned to those panes that specified a **floating** size. Once the sizes are determined, the origin for each pane relative to the panel origin is determined and entered into the **PANE** structure. A final pass is made over the list of panes in the current division, and, for each that is itself divided, the process is repeated.

If adjacent panes both have a border specified, the border space is shared between them.

If all of the panes have a fixed size and the total is less than the available space, there will be space that cannot be accessed by the application in the resulting structure.

If, after allocating space to the proportional panes, there is space remaining and no floating panes are in the current set, the remaining free space is allocated to the proportional panes.

The **ecdvpl** routine returns ERR and the structures are invalid for use by **ecdfpl** if one or more of the following occur:

The total size specified for fixed panes exceeds the space available.
The total fractions specified for the proportional panes exceed a total of 1.

The number of panes exceeds the number of positions available.

#include <cur05.h>
ecflin (*pane*, *firstline*, *firstcol*, *numlines*, *numcols*, **pat**, *xc*, **buf**, **mask)**
NLecflin (*pane*, *firstline*, *firstcol*, *numlines*, *numcols*, **pat**, *xc*, **buf**, *length*, **mask) char *pat, *buf, *mask;**
MBecflin (*pane*, *firstline*, *firstcol*, *numline*, *numcols*, *xc*, *buf*, *length*, *validcheck*, *validfunc***) int *validfunc;**

The **ecflin**, **NLecflin**, and **MBecflin** routines input field data to a pane. **ecflin** is retained to preserve traditional functionality. **NLceflin**, which supports code set pc850 only, is retained for backward compatibility, and **MBecflin** is provided for international character set support.

NLecflin works like **ecflin**, but has an additional parameter, *length*, which specifies the length of the buffer in which the input data is stored.

The **ecflin** routine inputs field data to the pane pointed to by the *pane* parameter. The *firstline* and the *firstcol* parameters specify the upper left corner of the field in the current window being shown in the pane. The *numcols* parameter specifies the number of columns in the field, and the *numlines* parameter specifies the number of rows in the field.

The **buf** parameter points to a buffer into which input data is stored. The buffer must be at least *numlines* ¦ *numcols* characters long.

The *xc* parameter specifies the first **NLSCHAR** to be entered into the field. If the *xc* parameter is a null character, it is ignored.

The **pat** and **mask** parameters specify the set of characters that are to be accepted as valid input.

The position in the field may not always correspond to the position in the input buffer. Input is accepted from the terminal as long as the cursor remains within the bounds of the field. However, if the input buffer is filled before the cursor exits the field, input processing stops and **ecflin** returns.

Cursor movement that moves the cursor outside the field is allowed and is reflected on the display. If cursor movement places the cursor in a position where data input would cause the input buffer to overflow, input processing stops. Any data keys entered are checked against the character set specified by the **pat** parameter. If the data character is acceptable, then it is echoed. If the character is not acceptable, then the **ecflin** routine returns its value.

Insert and delete keys are honored and data is shifted within the field as needed. If the field spans more than one line and insertions or deletions are made, then data that is shifted out of one line of the field is shifted into the end of the next line. Data shifted out of the field is lost. When characters are deleted, null characters are shifted into the end of the field.

The **pat** parameter points to a string that indicates the set of

characters that is acceptable as valid input. These characters include all code points of the P0 code page (see "display symbols" in topic 2.4.4). The string is formed of these codes:

U Uppercase letters: **'A'--'Z'** plus the accented uppercase letters from code page P0.
L Lowercase letters: **'a'--'z'** plus the accented lowercase letters from code page P0.
N Numeric characters: **'0'--'9'**.
A Alphanumeric characters: **'A'--'Z'**, **'a'--'z'**, and **'0'--'9'** plus the accented letters from code pages P0, P1, and P2.
B Blank (space character--0x20).
P Printable characters: **blank**--**'¬'** (0x20--0x7E).
G Graphic characters: **'!'--'¬'** (0x21--0x7E).
X Hexadecimal characters: **'0'--'9'**, **'A'--'F'**, and **'a'--'f'**.
C Control Characters:
Cursor Up, Cursor Down, Cursor Left, Cursor Right
Backspace
Back-tab (to first position of field)
Insert (enable or disable insert mode)
Delete (delete current character)
New-line (to left column and down one line)
D Default characters:
0x20--0x7E
0x80--0xFF
Controls, as defined for code **C**.
Z Application-specified character set
\+ Allows characters indicated by following codes.
\- Does not allow characters indicated by following codes.

If the first character of **pat** is **+** or **-**, the set of characters specified by the rest of the string is added to (**+**) or taken from (**-**) the default characters (which can also be specified with **D**). If the first character in this string is not **+** or **-**, then the set of characters specified by **pat** replaces the default. After the first character, the sets indicated are allowed unless preceded by a **-** (minus sign). For example:

"PC-L" Allows the printable and control characters, except for lowercase letters.
"-CBN" Allows all of the default characters, except for control characters, blanks, or numeric characters.

If the **pat** string contains a **Z**, then the array pointed to by the **mask** parameter specifies a character validity mask. This array must be exactly 64 bytes long (512 bits), where each bit corresponds to a character code as returned by **wgetch**. The bytes in the array correspond as follows:

Bytes 0--31 P0 characters 0x00--0xFF
Bytes 32--63 Keycodes 0x100--0x1FF

If a given bit is set to 1, then the corresponding character is accepted (for **+Z**) or rejected (for **-Z**). If a bit is set to 0, then the acceptance status of the corresponding character, as determined by the rest of **pat**, is not changed.

Upon successful completion, the code associated with the last input that terminated input is returned.

The **ecflin** routine returns ERR if one or more of the following are true:

There is an error in the parameters.
The *firstline* parameter is outside the window.
The *firstcol* parameter is outside the window.
The *numcols* parameter is too large.
The *numlines* parameter is too large.

The **MBecflin** routine works like the **NLecflin** routine, except that it also handles multiple byte and double display width character input. The definitions of the parameters **pane**, **firstline**, **firstcol**, **numline**, **numcols**, **xc**, **buf**, and **length** are the same as those described for **NLecflin**.

When **validcheck** is TRUE, **MBecflin** calls the routine **validfunc** (**c**) where **c** is the input code, and checks the return value from it. If the return value is TRUE, **c** is considered valid and processing continues; otherwise, **c** is considered invalid and processing terminates.

When **validcheck** is FALSE, **MBecflin** performs a default validity check in which any input data code not defined in the current locale is considered invalid (that is, **_mblen** returns a value of -1). **c** may be data code in **NLSCHAR** (**mbchar_t, file code**), or keypad code defined in **cur02.h**.

The routine **validfunc** must be defined by the user for the validity check, and a conversion from file code to wide code might be required in the routine to achieve code independence.

When an entered character overwrites half of an existing double display width character, the system replaces the remaining half of that existing character with a blank. If a character is inserted when the cursor is positioned at the second column of a double display width character, the cursor is adjusted to the first column of that character before insertion. A deletion, on the other hand, always deletes a whole character regardless of how many columns it occupies.

When a double display width character is entered at the last column of any line but the last line, the character enters the input buffer. On the screen, however, it is replaced by two partial-character indicators (@@), appearing at the last column of the current line, and at the beginning column of the next line respectively. Note that this is only a display-time feature. The original character is still kept as is in the input buffer. For example, during the editing session in the insertion mode, if there are insertions or deletions before that character, the character will be displayed normally whenever it is assigned two consecutive columns.

echo ( )
noecho ( )

The **echo** routine causes the terminal to echo characters to the display. If **echo** is set on, **wgetch** places all input into the data structure for the window.

The **noecho** routine turns **echo** off. If **echo** is turned off, characters are not written to the display.

#include <cur05.h>
ecpnin (*pane*, *boolf*, *xc***)**

The **ecpnin** routine causes the pane to accept keyboard input. The pane specified by the *pane* parameter is scrolled, if necessary, to ensure that the cursor is visible on the display. Keyboard input is then accepted. If the *boolf* parameter is TRUE and if the input character is a simple cursor movement, then the resulting cursor position is reflected on the display. Further input is then read from the terminal. If the *boolf* parameter is FALSE, or if the input character is not a simple cursor movement, then the value of the input character is returned.

The *xc* parameter specifies the first **NLSCHAR** to be assumed from the display. If *xc* is a null character, then it is ignored.

This routine tracks the locator cursor if locator tracking is enabled (see "trackloc" in topic 1.2.74.7).

void ecpnmodf (*pane***)**

The **ecpnmodf** macro marks the panel that contains the pane specified by the *pane* parameter as modified. This information is used by **ecrfpl** to determine whether a panel needs to be written to the display.

#include <cur05.h>
ecrfpl (*panel***)**

The **ecrfpl** routine refreshes the panel specified by the *panel* parameter. If that panel is partially obscured by other panels, then those panels are also written to the display. If the *panel* parameter is NULL, then all panels that have been marked as modified (with **ecpnmodf**) are written. If any panels have been removed (with **ecrmpl**), then all panels are written.

#include <cur05.h>
ecrfpn (*pane***)**

The **ecrfpn** routine refreshes the pane specified by the *pane* parameter on the display. If the pane is the active pane, then the window might be scrolled to ensure that the cursor is visible. If the pane is not active, then the window is not scrolled.

The **ecrfpn** routine always returns the value OK.

#include <cur05.h>
ecrlpl (*panel***)**

The **ecrlpl** routine returns the structures associated with the panel specified by the *panel* parameter to the free storage pool. This

includes all window structures associated with the panes of the panel, all Extended Curses private structures, and any added window structures. The panel and associated pane structures are not released and can be reused.

The **ecrlpl** routine always returns the value OK.

#include <cur05.h>
ecrmpl (*panel***)**

The **ecrmpl** routine removes the panel specified by the *panel* parameter from the list of panels that are currently being displayed. If the panel is not currently in that list, no action is taken and no error is returned. This routine should be followed by a call to **ecrfpl** to update the display.

The **ecrmpl** routine always returns the value OK.

#include <cur05.h>
ecscpn (*pane*, *numlines*, *numcols***)**

The **ecscpn** routine causes the pane specified by the *pane* parameter to be scrolled over the underlying window the distance indicated by the *numcols* and the *numlines* parameters. The *numcols* parameter specifies the distance to scroll horizontally and the *numlines* parameter specifies the distance to scroll vertically. These parameters can be positive or negative and may imply a movement that positions the viewport partially or completely off the window. If such a position results from the scroll, the scroll stops after moving as far in the indicated direction as possible. Positive values move to the right or down. Negative values move to the left or up.

If there are other panes linked to the pane specified, those panes will also scroll an amount necessary to maintain the identical horizontal or vertical positioning on the respective windows. If the resulting position requires placing the viewport partially or completely off the window, the scroll request terminates at the edge of the window.

#include <cur05.h>
ecshpl (*panel***)**

The **ecshpl** routine shows the panel specified by the *panel* parameter on the terminal.

If the specified panel is currently the top panel, no action is taken and no error is returned. If there is another top panel, the active pane in that panel is changed to the inactive state. The specified panel is placed at the top of the panel chain. This routine should be followed by a call to **ecrfpl** to update the display.

The **ecshpl** routine always returns the value OK.

#include <cur05.h>

ectitl (title, *line*, *col*)
char *title;

The **ectitl** routine creates or modifies the title panel. The title panel is always visible, that is, on top of any other panels. The **title** parameter points to a character string that is displayed as the new title. If **title** is NULL, then any existing title is removed. The *line* and *col* parameters specify the coordinates for the upper left corner of the title panel. If *firstline* is not valid, then it defaults to 1. If *firstcol* is not valid, then the title will be centered.

endwin ( )

The **endwin** routine ends window routines before exiting. Ending window routines before exiting restores the terminal to the state it was before **initscr** (or **gettmode** and **setterm**) was called. **endwin** should always be called before exiting. **endwin** does not exit.

erase ( )
werase (*win*)
perase (*pane*)

The **erase** routine clears the window and sets it to blanks without setting the clear flag. Similarly, **werase** erases the window specified by the **win** parameter, and **perase** erases the pane specified by the *pane* parameter. This is analogous to the **clear** routine, except that it does not cause a clear-screen sequence to be generated on a **refresh**.

extended (*boolf*)

The **extended** routine turns on and off the combining of input bytes into multibyte extended characters. If the *boolf* parameter is TRUE, then this input processing is turned on; if FALSE, then it is turned off. By default, **extended** processing is initially turned on.

flash ( )

The **flash** routine displays a visual bell on the terminal screen if one is available. If a visual bell is not available, then **flash** toggles the terminal speaker or bell.

The **flash** routine always returns the value OK.

fullbox (*win*, *vert*, *hor*, topl, topr, botl, botr)
NLSCHAR *vert*, *hor*, topl, topr, botl, botr;

The **fullbox** routine puts box characters on the edges of the window. The *vert* parameter specifies the **NLSCHAR** to use for the vertical sides. The *hor* parameter specifies the **NLSCHAR** to use for the horizontal lines. The **topl** and the **topr** parameters specify the **NLSCHAR**s to use for the top left and the top right corners. The **botl** and the **botr** parameters specify the **NLSCHAR**s to use for the bottom left and the bottom right corners.

The **fullbox** routine returns ERR if an attempt is made to scroll when **scrollok** is not active.

The **fullbox** routine does not accept double display width characters. If a double width character is used, the **fullbox** routine substitutes a single display character and draws the box. The system returns ERR.

The **fullbox** routine is a macro that invokes **superbox**.

#include <cur02.h>

NLSCHAR getch ( )
NLSCHAR wgetch (*win***)**
NLSCHAR mvgetch (*line*, *col***)**
NLSCHAR mvwgetch (*win*, *line*, *col***)**

The **getch** routine gets a character from the terminal and echoes it on the window, if necessary. If **noecho** has been set, then the window does not change. **noecho** and either **crmode** or **raw** must be set for Extended Curses to know what is actually on the terminal. If these settings are not correct, **wgetch** sets **noecho** and **crmode** and resets them to the original mode when done.

Upon completion, the NLSCHAR for the data key or one of the following values is returned:

KEY_NOKEY	**nodelay** is active and no data is available.
KEY_xxxx	**keypad** is active and a control key was recognized. See the **cur02.h** header file for a complete list of the key codes that can be returned.
ERR	Echoing the character would cause the screen to scroll illegally.

#include <cur02.h>

NLSCHAR getstr (*string***)**
NLSCHAR wgetstr (*win*, *string***)**
NLSCHAR mvgetstr (*line*, *col*, *string***)**
NLSCHAR mvwgetstr (*win*, *line*, *col*, *string***)**

The **getstr** routine gets a string through the window and stores it in the location pointed to by the *string* parameter. The string may contain single-shift control codes. The area pointed to must be large enough to hold the string. **getstr** calls **wgetch** to get the characters until a new-line character or some other control character is encountered.

Upon completion, one of the following values is returned:

OK	The input string was terminated with a new-line character.
KEY_NOKEY	**nodelay** is active and no data is available.
KEY_xxxx	The input string ended with a control key, and the code for this key was returned. See the **cur02.h** header file for a complete list of the key codes that

can be returned.

ERR The string caused the screen to scroll illegally.

gettmode ( )

The **gettmode** routine issues the needed control operation to the display device driver to save the processing flags in a fixed global area. **gettmode** is invoked by **initscr** and is not normally called directly by applications.

getyx (*win*, *line*, *col***)**

The **getyx** routine stores the current (*line, col*) coordinates of window specified by the *win* parameter into the variables *line* and *col*. Because **getyx** is a macro and not a subroutine, the names of *line* and *col* are passed, rather than their addresses.

Upon successful completion, *line* and *col* contain the current row and column coordinates for the cursor in the specified window.

NLSCHAR inch ( )
NLSCHAR winch (*win***)**
NLSCHAR mvinch (*line*, *col***)**
NLSCHAR mvwinch (*win*, *line*, *col***)**

The **inch** routine returns the **NLSCHAR** at the current (*line, col*) coordinates on the specified window. No changes are made to the window.

Upon successful completion, the code for the character located at the current cursor location is returned.

WINDOW *initscr ( )

The **initscr** routine performs screen initialization. **initscr** must be called before any of the screen routines are used. It initializes the terminal-type data, and without it, none of the Extended Curses routines can operate properly.

If standard input is not a **tty**, **initscr** sets the specifications to the terminal whose name is pointed to by **Def_term** (initially **"dumb"**). If the value of the **bool** global variable **My_term** is TRUE, **Def_term** is always used.

If standard input is a terminal, the specifications for the terminal named in the environment variable **TERM** are used. These specifications are obtained from the **terminfo** description file for that terminal.

The **initscr** routine creates the structures for **stdscr** and **curscr** and saves the pointers to those structures in global variables with the corresponding names.

Upon successful completion, a pointer to **stdscr** is returned.

insch (*xc***)**
winsch (*win*, *xc***)**
mvwinsch (*win*, *line*, *col*, *xc***)**
mvinsch (*line*, *col*, *xc***)**

The **insch** routine inserts the **NLSCHAR** specified by the *xc* parameter into the window at the current (*line, col*) coordinates. Each character after the inserted character shifts to the right and the last byte on the line disappears.

If the current position is at the second column of a double display width character, the position is moved left to the first byte of that character before the specified **NLSCHAR** is inserted.

If a double display width character is inserted at the last column of the window, nothing is done and ERR is returned; otherwise the **insch** routine always returns the value OK.

insertln ( )
winsertln (*win***)**

The **insertln** routine inserts a line above the current line. Each line below the current line is shifted down, and the bottom line disappears. The current line becomes blank and the current (*line, col*) coordinates remain unchanged.

The **insertln** routine always returns the value OK.

keypad (*boolf***)**

The **keypad** routine turns on and off the mapping of key sequences to single integers. If the *boolf* parameter is TRUE, input processing is turned on. If the *boolf* parameter is FALSE, input processing is turned off. By default, input processing is initially turned off.

When turned on, sequences of characters from the terminal are translated into integers that are defined in the **cur02.h** header file.

The codes available on a given terminal are determined by the **terminfo** terminal description file.

The **keypad** routine always returns the value OK.

leaveok (*win*, *boolf***)**

The **leaveok** routine sets a flag, used by the window specified by the *win* parameter, which controls where the cursor is placed after the window is refreshed. If the *boolf* parameter is TRUE, when the window is refreshed, the cursor is left at the last point where a change was made on the terminal, and the current (*line, col*) coordinates for the window specified by the *win* parameter are changed accordingly. If the (*line, col*) coordinates are outside the window, the coordinates are forced to (0, 0). If the *boolf* parameter is FALSE, when the window is refreshed, the cursor is moved to the current (*line, col*) coordinates within the window. The

controlling flag is initially set to FALSE.

The **leaveok** routine always returns the value OK.

char *longname ( )

The **longname** routine returns a pointer to a static area that contains the long (full) name of the terminal as it appears in the **terminfo** entry for the terminal.

meta ( )
nometa ( )

The **meta** routine prevents the stripping of the eighth bit of each keyed character.

The **nometa** routine causes the eighth or most-significant bit of each keyed character to be stripped. Not all terminals support the stripping of bits.

The **meta** and **nometa** routines always return the value OK.

move (*line*, *col***)**
wmove (*win*, *line*, *col***)**

The **move** routine changes the current (*line, col*) coordinates of the window to the coordinates specified by the *line* and *col* parameters.

The **move** routine returns ERR if the destination for the cursor is outside the window or viewport.

mvcur (*line*, *col*, **new***line*, **new***col***)**
int *line*, *col*, **new***line*, **new***col*;

The **mvcur** routine moves the terminal's cursor from the coordinates specified by the *line* and *col* parameters to the coordinates specified by the **newline** and **newcol** parameters. The **line** and **col** parameters must specify the current coordinates.

It is possible to use this optimization without the benefit of the screen routines. In fact, **mvcur** should not be used with the screen routines. Use **move** and **refresh** to move the cursor position and inform the screen routines of the move.

mvwin (*win*, *line*, *col***)**

The **mvwin** routine moves the position of the viewport or the subwindow specified by the *win* parameter from its current starting coordinates to the coordinates specified by the *line* and *col* parameters. The *line* parameter specifies the row on the display for the top row of the window. The *col* parameter specifies the column on the display for the first column of the window.

The **mvwin** routine returns ERR if a part of the window position is outside the bounds of the window on which the viewport is defined.

WINDOW *newview (*win***, *numlines*, *numcols***)**

The **newview** routine creates a new window that has the number of lines specified by the *numlines* parameter and the number of columns specified by the *numcols* parameter. The new window is a viewport of the window specified by the *win* parameter and starts at the current (*line, col*) coordinates of the window specified by the *win* parameter. The resulting window's initial position on the display is set to (0, 0).

The viewport window returned by **newview** is a special subwindow that is suitable for viewport scrolling. Viewport scrolling here refers to the type of scrolling that is characteristic of full-screen editors.

Because the returned viewport window is a subwindow, any change made in either window in the area covered by the viewport window appears in both windows. Both windows actually share the relevant storage area. A viewport window cannot be scrolled using **scroll**.

Other than the exceptions noted above, viewport windows behave like subwindows.

Upon successful completion, a pointer to the control structure for the new viewport is returned.

The **newview** routine returns ERR if the window specified by the *win* parameter is a subwindow or a viewport, or if sufficient storage is not available for the new structures.

WINDOW *newwin (*numlines***, *numcols*, *firstline*, *firstcol***)**

The **newwin** routine creates a new window that contains the number of lines specified by the *numlines* parameter and the number of columns specified by the *numcols* parameter. The new window starts at the coordinates specified by the *firstline* and the *firstcol* parameters.

If the *numlines* parameter is 0, that dimension is set to (**LINES** - *firstline*). If the *numcols* parameter is 0, that dimension is set to (**COLS** - *firstcol*). Therefore, to get a new window of dimensions (**LINES** ¦ **COLS**), use:

```
newwin (0, 0, 0, 0)
```

The size specified for the window can exceed the size of the real display. In this case, a viewport or subwindow must be used to present the data from the window on the terminal.

Upon successful completion, a pointer to the new window structure is returned.

The **newwin** routine returns ERR if any of the parameters are invalid, or if there is insufficient storage available for the new structure.

nl ( )
nonl ( )

The **nl** routine sets the terminal to ***nl mode***. When in nl mode, the system maps **'\r'** (return characters) to **'\n'** (new-line or line-feed characters). If the mapping is not done, **refresh** can do more optimization. **nonl** turns nl mode off.

The **nl** routine and **nonl** do not affect the way in which **waddch** processes new-line characters.

The **nl** and **nonl** routines always return the value OK.

nodelay (*boolf***)**

The **nodelay** routine controls whether read requests wait for input if no keystroke is available. If the *boolf* parameter is FALSE, then the read routines wait for operator input. This is the default setting. If the *boolf* parameter is TRUE, then the read routines return immediately if no keyboard data is available.

If **nodelay** is set (TRUE) and if no keystroke is available from the keyboard, then **getch** returns **KEY_NOKEY**, which is defined in the **cur02.h** header file.

The **nodelay** routine always returns the value OK.

overlay (*win1***, *win2***)**

The **overlay** routine overlays the window specified by the *win1* parameter on the window specified by the *win2* parameter. The contents of the window specified by the *win1* parameter, insofar as they fit, are placed on the window specified by the *win2* parameter at their starting (*line, col*) coordinates. This is done nondestructively; that is, blanks on the *win1* window leave the contents of the space on the *win2* window untouched.

The **overlay** routine moves data only if the data is nonblank or if the display attribute is different.

The only data that is considered for moving from the *win1* window to the *win2* window is data that occupies display positions that are common to both windows.

The **overlay** routine is implemented as a macro that invokes **overput**, which uses **waddch** to transfer the data from window to window.

The **overlay** routine returns ERR if an attempt is made to write to the lower right corner of the display and **scrollok** is FALSE.

overwrite (*win1***, *win2***)**

The **overwrite** routine copies data from the window specified by the *win1* parameter to the window specified by the *win2* parameter. The contents of the *win1* window, insofar as they fit, are placed on the *win2* window at their starting (*line, col*) coordinates. This is done destructively; that is, blanks on the *win1* window become blanks on the *win2* window.

Only the data that occupies positions on the display that are common to the two windows is moved from the *win1* window to the *win2* window.

The **overwrite** routine is implemented as a macro that invokes **overput** which uses **waddch** to transfer the data from window to window.

The **overwrite** routine returns ERR if an attempt is made to write to the lower right corner and **scrollok** is FALSE.

printw (fmt [**, value,...])**
wprintw (*win***, fmt** [**, value,...])**
char *fmt;

The **printw** routine performs a **printf** on the window using the format control string specified by the **fmt** parameter and the values specified by the **value** parameters. The output to the window starts at the current (*line, col*) coordinates. Use the field width options of **printf** to avoid leaving things on the window from earlier calls. See "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf" in topic 1.2.208 for details.

The **printw** routine returns ERR if it causes the screen to scroll illegally.

raw ( )
noraw ( )

The **raw** routine sets the terminal to raw mode. In raw mode, canonical processing by the device driver and signal processing are turned off. The **noraw** routine turns off raw mode.

The **raw** and **noraw** routines always return the value OK.

refresh ( )
wrefresh (*win***)**

The **refresh** routine synchronizes the terminal screen with the window. If the window is not a screen, then only the part of the display covered by it is updated. **refresh** checks for possible scroll errors at display time.

The **refresh** routine returns ERR if the change specified is in the last position of a window that includes the lower right corner of the display, or if it would cause the screen to scroll illegally. If it would cause the screen to scroll illegally, **refresh** updates whatever can be updated without causing the scroll.

resetty (*boolf***)**

The **resetty** routine restores the terminal status flags that were previously saved by **savetty**. If the *boolf* parameter is TRUE, then the screen is cleared in addition to resetting the terminal. **resetty** is performed automatically by **endwin** and is not normally called directly by applications.

savetty ( )

The **savetty** routine saves the current terminal status flags. **savetty** is performed automatically by **initscr** and is not normally called directly by applications.

scanw (fmt [**, pointer,...])**
wscanw (*win***, fmt** [**, pointer,...])**
char *fmt;

The **scanw** routine performs a **scanf** through the window using the format control string specified by the **fmt** parameter. **scanw** uses **wgetstr** to obtain the string, then invokes the internal routine for **scanf** to process the data. See "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf, wsscanf" in topic 1.2.241 for details.

scroll (*win***)**

The **scroll** routine moves the data in the window specified by the *win* parameter up one line and inserts a new blank line at the bottom.

scrollok (*win***,** *boolf***)**

The **scrollok** routine sets the scroll flag for the window specified by the *win* parameter. If the *boolf* parameter is TRUE, then scrolling is allowed. The default setting is FALSE, which prevents scrolling.

sel_attr (set)
int *set;

The **sel_attr** routine allows you to change the selection and priority of attributes for the run-time terminal. The **set** parameter points to a null-terminated integer array that contains display attribute values from the **cur03.h** header file in the order that you want them regardless of whether or not they are available on the terminal.

Groups of attributes (colors and fonts) cannot be split in the array. For instance, all foreground colors specified must be in adjacent locations in the array.

The first element of a group of attributes must be the default color or font of the terminal. For example, the first foreground color specified is usually **F_WHITE**, and the first background color specified is usually **B_BLACK**.

It is recommended that **sel_attr** only be called before **initscr**. If **sel_attr** is called after **initscr**, then the routine **setup_attr** should be called after calling **sel_attr**. If **sel_attr** is called after data has been added to a window, the values in the associated attribute array for that window may denote different attributes than the original attributes used when displaying the data (except **NORMAL** which remains constant). A subsequent refresh of the window shows the different attributes only if the data has been modified or if a total refresh has been forced by a previous call to **touchwin**.

The **sel_attr** routine always returns the value OK.

setterm (**name**)
char *name;

The **setterm** routine sets the terminal characteristics to be those of the terminal specified by the **name** parameter. **setterm** is called by **initscr** so you do not normally have to use it unless you wish to use just the cursor motion optimizations.

setup_attr ()

The **setup_attr** routine creates the display attribute masks assigned to the attribute variables declared in the **cur01.h** header file. The priorities of the attributes determine how the masks are created.

This routine is called by **initscr** and is not normally called by applications. This routine should only be called following a call to **sel_attr** which follows a call to **initscr**.

standend ( )
wstandend (*win***)**

The **standend** routine stops displaying characters in standout mode.

standout ( )
wstandout (*win***)**

The **standout** routine starts displaying characters in standout mode. Any characters added to the window are put in standout mode on the terminal if the terminal has that capability. The first available attribute as determined by **sel_attr** is used for standout. This is normally the reverse attribute when the default display attribute priority is used.

The **standout** routine always returns the value OK.

WINDOW *subwin (*win***,** *numlines***,** *numcols***,** *firstline***,** *firstcol***)**

The **subwin** routine creates a subwindow in the window pointed to by the *win* parameter. The subwindow has the number of lines specified by the *numlines* parameter and the number of columns specified by the *numcols* parameter. The new subwindow starts at the coordinates specified by the *firstline* and the *firstcol* parameters. Any change made to the window or the subwindow in the area covered by the subwindow is made to both windows.

The *firstline* and *firstcol* parameters are specified relative to the overall screen, not to the relative (0, 0) of the window specified by the *win* parameter.

If the *numlines* parameter is 0, then the lines dimension is set to (**LINES** - *firstline*). If the *numcols* parameter is 0, then the column dimension is set to (**COLS** - *firstcol*).

Upon successful completion, a pointer to the control structure for the new subwindow is returned.

The **subwin** routine returns ERR if the window specified by the *win* parameter already has a subwindow, or if there is insufficient storage for the new control structure.

superbox (*win*, *line*, *col*, *numlines*, *numcols*, *System V*, *hor*, **topl**, **topr**, **botl**, **NLSCHAR** *System V*, *hor*, **topl**, **topr**, **botl**, **botr;**

The **superbox** routine draws a box on the window specified by the *win* parameter. The *line* and *col* parameters specify the starting coordinates for the box. The *numlines* parameter specifies the depth of the box. The *numcols* parameter specifies the width of the box. The *System V* parameter specifies the **NLSCHAR** to use for vertical delimiting. The *hor* parameter specifies the **NLSCHAR** to use for horizontal delimiting. The **topl**, **topr**, **botl**, and **botr** parameters specify the **NLSCHAR**s to use for the top left corner, the top right corner, the bottom left corner, and the bottom right corner, respectively.

If the window specified by the *win* parameter is a **_SCROLLWIN** window and scrolling is not allowed, then the bottom right corner is not put on the window.

The **superbox** routine uses **addch** to place the characters on the window.

The **superbox** routine returns ERR if the defined box is outside the window, or an attempt is made to write to the lower right corner of the display when **scrollok** is off.

touchwin (*win***)**

The **touchwin** routine makes it appear as if every location on the window specified by the *win* parameter has been changed. This is useful when overlapping windows are to be refreshed. A subsequent **refresh** request considers all portions of the window as potentially modified. If **touchwin** is not used, then only those positions of the window that have been addressed by an **addch** are inspected.

trackloc (*boolf***)**

The **trackloc** routine turns on and off the tracking of the locator cursor on the screen. If the *boolf* parameter is TRUE, then locator tracking is turned on; if FALSE, then it is turned off. By default, locator tracking is initially turned on.

The keycode **KEY_LOCESC** is returned from **getch** when a locator report is input. The locator report is stored in the global **char** array **ESCSTR**, which is 128 bytes long.

Locator tracking is handled by the **ecpnin** routine.

tstp ( )

The **tstp** routine saves the current **tty** state and then put the process to sleep. When the process is restarted, the **tty** state is restored and then **wrefresh (curscr)** is called to redraw the screen. **initscr** sets the signal **SIGTSTP** to trap **tstp**.

The **tstp** routine always returns the value OK.

#include <cur04.h>

char *unctrl (*ch***)**

The **unctrl** routine returns a string that represents the value of the *ch* parameter. Control characters become the lowercase equivalents preceded by a ^ (circumflex). Other letters are unchanged. This function supports only the P0 characters 0x00 through 0x7F.

Upon successful completion, a pointer to the string for the parameter character is returned.

vscroll (*win***, *numlines*, *numcols***)**

The **vscroll** routine scrolls the viewport specified by the *win* parameter on the window.

The *numlines* parameter specifies the direction and amount to scroll up or down. If the *numlines* parameter is positive, the viewport scrolls down the number of lines specified. If the *numlines* parameter is negative, the viewport scrolls up the number of lines specified.

The *numcols* parameter specifies the direction and amount to scroll left or right. If the *numcols* parameter is positive, the viewport scrolls to the right the number of characters specified. If the *numcols* parameter is negative, then the viewport scrolls to the left the number of characters specified.

The **vscroll** routine always scrolls as much of a requested scroll as possible. Specifying a parameter with a magnitude larger than that of the underlying window is not an error.

The **vscroll** routine calls **touchwin** if any scrolling is done.

The **vscroll** routine returns ERR if the window specified by the *win* parameter is not a window created by a call to **newview**.

File

/usr/lib/terminfo/?/* Compiled terminal capability data base.

Related Information
In this book: "curses" in topic 1.2.56 and "terminfo" in topic 2.3.59.

The discussion of Extended Curses in *AIX Programming Tools and Interfaces*.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.75 fabort

Purpose
Aborts or undoes file changes.

Syntax

int fabort (fildes)
int fildes

Description
The **fabort** system call aborts data changes made to the file specified by the **fildes** parameter. The file must be a regular file, not a pipe or special file, and the file descriptor must either be open for write and defer commit (**O_WRONLY** | **O_DEFERC**) or read/write and defer commit (**O_RDWR** | **O_DEFERC**). If the file has not changed since it was last committed (see "fsync, fcommit" in topic 1.2.87), the **fabort** system call has no effect.

The file's content is restored to the state of the file after the last **fsync** system call. Changes made to the file's content since the last **fsync** are undone. Changes made to the file's mode, owner, or access and modification times are not undone, however, by this system call. The file remains open so further changes may be made.

Note: The file offset pointer for this and any other of the file's open file descriptors are unchanged by **fabort** and actually may point beyond the end of the file since the file size is reset.

If the file is newly created (not just truncated), **fabort** undoes all of the changes to the file, leaving a zero-length file, but does not undo the file creation.

If the file is open for writing in defer-commit mode by more than one process, **fabort** undoes the changes made to the file by all of the processes since the last **fsync** performed by any one of the processes. Also, since commit operations implied by the **close** system call are not done until the last close of the file, **fabort** undoes any changes between the last **fsync** and a **close** that is not the last close of the file. The use of the **lockf** system call is recommended to coordinate multiple writers.

Warning: If the file is opened in defer-commit mode by some processes and opened for write, but not in defer-commit mode, by other processes, the results of the **fabort** system call are undefined.

Return Value
Upon successful completion, a value of 0 is returned. If the **fabort** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **fabort** system call fails and the file is not modified if one or more the following are true:

EAGAIN Another process holds an enforced mode record lock on this file.

EBADF The **fildes** parameter is not a valid file descriptor for a regular file.

EBADF The **fildes** parameter does not specify an open file.

EIO An I/O error occurred. The file content is still rolled back.

ENOSPC The file system ran out of space. The file content is still rolled back.

ESITEDN1 The operation could not be done because contact with the storage site is lost. The file content is still rolled back unless the problem is only a network communication problem and there are other processes with this file open for writing which have not lost contact with the storage site.

ESITEDN2 The operation was terminated because a site failed.

Related Information
In this book: "chmod, fchmod" in topic 1.2.44, "chown, fchown" in topic 1.2.45, "close, closex" in topic 1.2.48, "exit, _exit" in topic 1.2.73, "fsync, fcommit" in topic 1.2.87, "fcntl, flock, lockf" in topic 1.2.78, "open, openx, creat" in topic 1.2.199, and "utime" in topic 1.2.321.

1.2.76 fclear

Purpose
Clears space in a file, freeing unused disk space.

Syntax

```
long fclear (fildes, nbytes)
int fildes;
unsigned long nbytes;
```

Description
The **fclear** system call zeros the number of bytes specified by the **nbytes** parameter starting at the current position of the file open on file descriptor **fildes**. This function differs from the logically equivalent write operation in that it returns full blocks of binary zeros to the file system, constructing ***holes*** in the file. The seek pointer of the file is advanced by **nbytes**.

If you **fclear** past the end of a file, the rest of the file is cleared, and the seek pointer is advanced by **nbytes**. The file size is updated to include this new hole, which leaves the current file position at the byte immediately beyond the new end-of-file. Successful completion of the **fclear** system call clears the set-user-ID and set-group-ID attributes of the file.

Return Value
Upon successful completion, a value of **nbytes** is returned. If the **fclear** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **fclear** system call fails if one or more of the following are true:

EIO I/O error.

EBADF The **fildes** option is not a valid file descriptor open for writing.

EINVAL The file is a FIFO, directory, or special file.

EAGAIN The write operation in **fclear** failed, due to an enforced write lock on the file.

If the Transparent Computing Facility is installed on your system, **fclear** can also fail if one or more of the following are true:

ESITEDN1 The storage site is down or has gone down since this file descriptor was issued.

ESITEDN2 The operation was terminated because a site failed.

Related Information
In this book: "ftruncate, truncate" in topic 1.2.88.

1.2.77 fclose, fflush

Purpose
Closes or flushes a stream.

Library
Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>


int fclose (stream)          int fflush (stream)
FILE *stream;                FILE *stream;
```

Description
The **fclose** subroutine writes buffered data to the stream specified by the **stream** parameter and then closes the stream.

The **fclose** subroutine is automatically called for all open files when the **exit** system call is invoked.

The **fflush** subroutine writes any buffered data for the stream specified by the **stream** parameter and leaves the stream open.

Return Value
Upon successful completion, both the **fclose** and the **fflush** subroutines return a value of 0. If either of these subroutines fails for any reason, it returns the value EOF.

Error Conditions
The **fclose** and **fflush** subroutines fail if one or more of the following are true:

EAGAIN The **O_NONBLOCK** flag is set for the file descriptor underlying **stream** and the process is delayed in the write operation.

EBADF The file descriptor underlying **stream** is not valid.

EFBIG An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

EINTR The function was interrupted by a signal.

EIO The process is a member of a background process group attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking **SIGTTOU** and the process group of the process is orphaned.

ENOSPEC There was no free space remaining on the device containing the file.

EPIPE An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal is also sent to the process.

Related Information
In this book: "close, closex" in topic 1.2.48, "exit, _exit" in

topic 1.2.73, "fopen, freopen, fdopen" in topic 1.2.82, "setbuf, setvbuf" in topic 1.2.247, and " stdio" in topic 1.2.283.

1.2.78 fcntl, flock, lockf

Purpose
Controls open file and socket descriptors.

Syntax

```
#include <unistd.h>

int (lockf, (d, cmd, arg)
int d, cmd, arg;
```

Description
The **fcntl** system call performs controlling operations on the open file or socket specified by the **d** parameter.

The **d** parameter is an open file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call, or a socket descriptor from a **socket** or **socketpair** system call. The **arg** parameter is a variable that depends on the value of the **cmd** parameter.

The following **cmd**s get a descriptor or associated flags or set those flags:

F_DUPFD Returns a new descriptor as follows:

Lowest numbered available descriptor greater than or equal to **arg**

Same object reference as the original descriptor

Same file pointer as the original file (that is, both file descriptors share one file pointer)

Same access mode (read, write or read/write)

Same locks

Same file status flags (that is, both file descriptors share the same file status flags)

The **close-on-exec** flag associated with the new descriptor is set to remain open across **exec** system calls.

F_GETFD Gets the flags of the descriptor **d**.

F_SETFD Sets the **close-on-exec** flag associated with the **d** parameter to the value of the low-order bit of **arg** (0 or 1 as for **F_GETFD**).

F_GETFL Gets the file status flags of the file descriptor **d**.

F_SETFL Sets the file status flags to the value of the **arg** parameter. Only the flags **O_NDELAY**, **O_NONBLOCK**, **O_APPEND**, and **O_ASYNC** can be set. If you attempt to set any other flags with **F_SETFL**, the **fcntl** system call does not set the flags and returns without an error message. **O_ASYNC** enables the **SIGIO** signal to be sent to the process or process group specified by the **F_SETOWN** when I/O is possible.

F_GETOWN Gets the process ID or process group ID set to receive **SIGIO** and

SIGURG signals; process group IDs are returned as negative values.

F_SETOWN Sets the process ID or process group ID to receive **SIGIO** and **SIGURG** signals. A process group ID is specified by giving **arg** as a negative value; a positive value is a process ID.

When using the file locking and unlocking **cmd**s (**F_GETLK**, **F_SETLK**, and **F_SETLKW**), the **arg** parameter is a pointer to a structure of type **flock**. The **flock** structure pointed to by the **arg** parameter describes the lock and is defined in the **fcntl.h** header file. It contains the following members:

```
short l_type;           /* F_RDLCK, F_WRLCK, F_UNLCK */
short l_whence;         /* flag for starting offset */
long  l_start;          /* relative offset in bytes */
long  l_len;            /* if 0 then until EOF */
unsigned long l_sysid;  /* node ID */
pid_t l_pid;            /* returned with F_GETLK */
```

l_type Describes the type of lock. Possible values are **F_RDLCK**, **F_WRLCK**, and **F_UNLCK**.

l_whence Defines the starting point of the relative offset, **l_start**. A value of 0 indicates the start of the file, 1 selects the current position, and 2 indicates the end of the file.

l_start Defines the relative offset in bytes, measured from the starting point in **l_whence**.

l_len Specifies the number of consecutive bytes to be locked.

l_sysid Contains the ID of the node that already has a lock placed on the area defined by the **fcntl** system call. This field is returned only when the **F_GETLK cmd** is used.

l_pid Contains the ID of a process that already has a lock placed on the area defined by the **fcntl** system call. This field is returned only when the **F_GETLK cmd** is used.

The following **cmd**s use the **flock** structure and perform operations associated with file locks:

F_GETLK Gets the first lock that blocks the lock described in the **flock** structure pointed to by **arg**. If a lock is found, the retrieved information overwrites the information in this structure. If no lock is found that would prevent this lock from being created, then the structure is passed back unchanged except that the lock type is set to **F_UNLCK**.

F_SETLK Sets or clears a file lock according to the **flock** structure pointed to by **arg**. **F_SETLK** is used to establish read (**F_RDLCK**) and write (**F_WRLCK**) locks, as well as to remove either type of lock (**F_UNLCK**). **F_RDLCK**, **F_WRLCK**, and **F_UNLCK** are defined by the **fcntl.h** header file. If a read or write lock cannot be set, **fcntl** returns immediately with an error value of -1.

F_SETLKW Works like **F_SETLK** except that if a read or write lock is blocked by existing locks, the process sleeps until the section of the file is free to be locked.

When a read lock has been set on a section of a file, other processes may also set read locks on that section or subsets of it. A read lock prevents any other process from setting a write lock on any part of the protected area. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any other process from setting a read lock or a write lock on any part of the protected area. Only one write lock and no read locks may exist for a specific section of a file at any time. The file descriptor on which a write lock is being placed must have been opened with write access.

Locks may start and extend beyond the current end of a file but may not be negative relative to the beginning of the file. A lock may be set to extend to the end of the file by setting **l_len** to 0. If such a lock also has **l_start** and **l_whence** set to 0, the whole file is locked.

Some general rules about file locking include:

Changing or unlocking part of a file in the middle of a locked sectio leaves two smaller sections locked at each end of the originally locked section.

When the calling process holds a lock on a file, that lock is replace by later calls to **fcntl**.

All locks associated with a file for a given process are removed whe a file descriptor for that file is closed by the process or the process holding the file descriptor ends.

Locks are not inherited by a child process after executing a **fork** system call.

If the Transparent Computing Facility is installed, the following also applies to use of **fcntl**:

If the file being locked is a replicated file, the primary copy mus be available, even if the file is open for read and another copy can otherwise still be used. This restriction is necessary to guarantee that no two processes on sites in different network partitions hold conflicting locks on the same file at the same time.

The loss of the site storing the file (storing the primary copy, i the file is replicated) causes a process that holds locks on the file to lose those locks. If not caught or ignored, this signal causes the process to be terminated.

Notes:

1. In addition to **fcntl**, the **lockf** system call can also be used to set write (exclusive) locks.

2. Deadlocks due to file locks in a distributed system are not always detected. When such deadlocks are possible, the programs requesting the locks should set timeout timers.

Compatibility Interfaces

The following additional interfaces are provided:

lockf (*fildes*, **F_LOCK**, *size*) is equivalent to **fcntl** (*fildes*, **F_SETLKW**, *flock*) where *flock* has:

l_type = **F_WRLCK**
l_whence = 1
l_start = (*size* >= 0) ? 0 : *size*
l_len = (*size* >= 0) ? *size* : -*size;*

lockf (*fildes*, **F_TLOCK**, *size*) is equivalent to **fcntl** (*fildes*, **F_SETLK**, *flock*) where *flock* is as for **F_LOCK:**

l_type = **F_WRLCK**
l_whence = 1
l_start = (*size* >= 0) ? 0 : *size*
l_len = (*size* >= 0) ? *size* : -*size;*

lockf (*fildes*, **F_ULOCK**, *size*) is equivalent to **fcntl** (*fildes*, **F_SETLK**, *flock*) where *flock* is as for **F_LOCK**, except **l_type** is **F_UNLCK:**

l_type = **F_UNLCK**
l_whence = 1
l_start = (*size* >= 0) ? 0 : *size*
l_len = (*size* >= 0) ? *size* : -*size;*

lockf (*fildes*, **F_TEST**, *size*) is equivalent to **fcntl** (*fildes*, **F_GETLCK**, *flock*) where *flock* is as for **F_LOCK:**

l_whence = 1
l_start = (*size* >= 0) ? 0 : *size*
l_len = (*size* >= 0) ? *size* : -*size;*
except that the return value is established as:

- if no conflicting lock exists, **lockf** returns 0.
- if a conflicting lock exists, **lockf** returns -1 and sets **errno** to EAGAIN.

The commands, **F_LOCK, F_TLOCK, F_ULOCK,** and **F_TEST** are defined in <**sys/lockf.h**>.

For 4.3BSD compatibility, the **flock** interface is also supported. To use **flock**, compile with the Berkeley Compatibility Library (**libbsd.a**).

flock (*fd, operation*)

where *operation* is the inclusive OR of **LOCK_SH, LOCK_EX** and, possibly, **LOCK_NB**, or **LOCK_UN**, is equivalent to **fcntl** (*fildes*, *cmd*, *flock*)

where *cmd* is **F_SETLK** (or, **F_SETLKW** if **LOCK_NB** is set) and *flock* is set follows:

l_type = **F_RDLCK** if **LOCK_SH**.
l_type = **F_WRLCK** if **LOCK_EX**.
l_type = **F_UNLCK** if **LOCK_UN**.
l_whence = 0
l_start = 0
l_len = 0

Return Value
Upon successful completion, the value returned depends on the value of the **cmd** parameter as follows:

cmd	Return Value
F_DUPFD	A new descriptor
F_GETFD	The value of the flag (only the low-order bit is defined)
F_GETLK	A value other than -1
F_SETFD	A value other than -1
F_GETFL	The value of file flags
F_SETFL	A value other than -1
F_SETLK	A value other than -1
F_SETLKW	A value other than -1
F_GETOWN	A process ID or a negative process group ID. Process group IDs are never 1
F_SETOWN	A value other than -1.

If the **fcntl** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **fcntl** system call fails if one or more of the following are true:

EBADF The **d** parameter is not a valid open file descriptor.

EBADF A read lock (**F_RDLCK**) is attempted on a file open only for writing (**O_WRONLY**) or a write lock (**F_WRLCK**) is attempted on a file open only for reading (**O_RDONLY**).

EMFILE The **cmd** parameter is **F_DUPFD** and 200 file descriptors are currently open.

EACCES The **cmd** parameter is **F_SETLK**, the **l_type** parameter is **F_RDLCK**, and the segment of the file to be locked is already write-locked by another process.

EACCES The **cmd** parameter is **F_SETLK**, the **l_type** parameter is **F_WRLCK**, and the segment of a file to be locked is already read-locked or write-locked by another process.

Note: Because in the future **errno** may be set to EAGAIN rather than to EACCES for the two errors described above, programs should expect and test for both values.

EDEADLK The **cmd** parameter is **F_SETLKW**, the lock is blocked by some lock from another process. Putting the calling process to sleep while waiting for that lock to become free would cause a deadlock.

ENOLCK The **cmd** parameter is **F_SETLK** or **F_SETLKW**, the type of lock is **F_RDLCK** or **F_WRLCK**, and there are no more file locks available. (Too many segments are already locked.)

EINVAL The **cmd** parameter is **F_GETLK**, **F_SETLK**, or **F_SETLKW** and the **arg** parameter or the data it points to is not valid.

EINVAL The **cmd** parameter is **F_DUPFD** and the **arg** parameter is negative or greater than 199.

If the Transparent Computing Facility is installed on your system, **fcntl** can also fail if one or more of the following are true:

ESITEDN1 The site which stores this file is now or has been down.

ESITEDN2 The operation was terminated because a site failed.

ETXTBSY An attempt was made to lock a region of a file which is currently being executed. This error is only for replicated files and only in situations where obtaining a lock at the primary copy of a replicated file system would interfere with processes which are currently executing this program.

ENOSTORE An attempt was made to lock a replicated file which has already been deleted.

EROFS A lock is requested for a file in a replicated file system in which the primary copy is unavailable.

ENFILE The system inode table at the current synchronization site is full.

Related Information
In this book: "close, closex" in topic 1.2.48, "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, "fcntl, flock, lockf," "open, openx, creat" in topic 1.2.199, and "fcntl.h" in topic 2.4.8.

1.2.79 feof, ferror, clearerr, fileno

Purpose
Checks the status of a stream.

Library
Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>

int feof (stream)                    void clearerr (stream)
FILE *stream;                        FILE *stream;

int ferror (stream)                  int fileno (stream)
FILE *stream;                        FILE *stream;
```

Description
These macros inquire about the status of a stream. Each of these macros also can be found in **libc.a** as subroutines.

The **feof** macro inquires about end-of-file. If EOF has previously been detected reading the input stream specified by the **stream** parameter, a nonzero value is returned. Otherwise, a value of 0 is returned.

The **ferror** macro inquires about input/output errors. If an I/O error has previously occurred when reading from or writing to the stream specified by the **stream** parameter, a nonzero value is returned. Otherwise, a value of 0 is returned.

The **clearerr** macro resets the error indicator and the EOF indicator to 0 for the stream specified by the **stream** parameter.

The **fileno** macro returns the integer file descriptor associated with the input pointed to by the **stream** parameter.

Note: Since these routines are implemented as macros, they cannot be declared or redeclared without first undefining the macro (after including **stdio.h**).

Error Conditions
The **feof**, **ferror**, and **fileno** subroutines fail if one or more of the following are true:

EBADF The file descriptor underlying **stream** is not valid.

EBADF The **stream** argument is not a valid stream.

Related Information
In this book: "fopen, freopen, fdopen" in topic 1.2.82, "open, openx, creat" in topic 1.2.199, and " stdio" in topic 1.2.283.

1.2.80 finite, logb, scalb

Purpose
Used in floating point calculations.

Library
Standard C Library (**libm.a**)

Syntax

```
#include <math.h>

int finite (x)
double x;

double logb (x)
double x;

double scalb (x, n)
double x;
int n;
```

Description
These subroutines are required for, or recommended by the IEEE standard 754 for floating point arithmetic.

finite(x) = 1 only when -infinity < **x** < +infinity.
finite(x) = 0 otherwise (when |**x**| = infinity or **x** is NaN ().

logb(x) returns **x** exponent **n**, a signed integer converted to double-precision floating point. It is set to 1 < |**x**|/2**n <2 unless **x**=0 or |**x**|=infinity.

scalb(x, n) = **x***(2****n**) computed, for integer **n**, without first computing 2****n**.

Related Information
In this book: "floor, ceil, fmod, fabs, rint" in topic 1.2.81, and "math.h" in topic 2.4.13.

1.2.81 floor, ceil, fmod, fabs, rint

Purpose
Computes floor, ceiling, remainder, and absolute value functions.

Library
Math Library (**libm.a**)

Syntax

```
#include <math.h>

double floor (x)                    double fmod (x, y)
double x;                           double x, y;

double ceil (x)                     double fabs (x)
double x;                           double x;
#include <stdlib.h>
#include <limits.h>

double rint (x)
double x;
```

Description
The **floor** subroutine returns the largest integer (as a **double**) not greater than the **x** parameter.

The **ceil** subroutine returns the smallest integer not less than the **x** parameter.

The **fmod** subroutine returns the remainder of **x** ÷ **y**. More precisely, this value is **x** if the **y** parameter is 0. Otherwise, it is the number **f** with the same sign as **x** such that **x** = **iy** + **f** for some integer **i**, and |**f**| < |**y**|.

The **fabs** subroutine returns the absolute value of **x**, |**x**|.

The **rint** subroutine returns one of the two nearest floating point integers to **x**. Which integer is returned is determined by the current floating point rounding mode.

If the current rounding mode is round toward -&infinity., then **rint(x)** is identical to **floor(x)**.

If the current rounding mode is round toward +&infinity., then **rint(x)** is identical to **ceil(x)**.

If the current rounding mode is round to nearest, then **rint(x)** rounds to the nearer of the two nearest floating point integers.

If the current rounding mode is round toward zero, then **rint(x)** is equivalent to truncating the fractional bits of x.

Note: The default floating point rounding mode is round towards zero.

Error Conditions
The **floor**, **ceil**, **fmod**, and **fabs** subroutines fail if one or more of the following are true:

EDOM The value of **x** is NaN.

EDOM The argument **y** is zero or one of the arguments is NaN.

ERANGE The result would cause an overflow.

Related Information
In this book: "abs" in topic 1.2.8.

1.2.82 fopen, freopen, fdopen

Purpose
Opens a stream.

Library
Standard I/O Library (**libc.a**)

Syntax
#include <stdio.h>

```
FILE *fopen (path, type)        FILE *fdopen (fildes, type)
char *path, *type;              int fildes;
                                char *type;
FILE *freopen (path, type, stream)
char *path, *type;
FILE *stream;
```

Description
The **fopen** subroutine opens the file named by the **path** parameter and associates a stream with it. **fopen** returns a pointer to the **FILE** structure of this stream.

The **path** parameter points to a character string that contains the name of the file to be opened.

The **type** parameter points to a character string that has one of the following values:

"r"	Open the file for reading
"w"	Truncate or create a new file for writing
"a"	Append (open for writing at end of file, or create for writing)
"r+"	Open for update (reading and writing)
"w+"	Truncate or create for update
"a+"	Append (open or create for update at end of file)

The **freopen** subroutine substitutes the named file in place of the open **stream**. The original **stream** is closed whether or not the **open** succeeds. **freopen** returns a pointer to the **FILE** structure associated with **stream**. The **freopen** subroutine is typically used to attach the pre-opened **streams** associated with **stdin**, **stdout**, and **stderr** to other files.

The **fdopen** subroutine associates a stream with a file descriptor obtained from an **open**, **dup**, **creat**, or **pipe** system call. These system calls open files but do not return pointers to **FILE** structures. Many of the standard I/O library subroutines require pointers to **FILE** structures. Note that the **type** of stream specified must agree with the mode of the open file.

When you open a file for update, you can perform both input and output operations on the resulting stream. However, an output operation cannot be directly followed by an input operation without an intervening **fseek** or **rewind**. Also, an input operation cannot be directly followed by an output operation without an intervening **fseek**, **rewind**, or an input operation that encounters the end of the file.

When you open a file for append (that is, when **type** is **"a"** or **"a+"**), it is impossible to overwrite information already in the file. You can use **fseek** to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is ignored. All

output is written at the end of the file and causes the file pointer to be repositioned to the end of the output.

If two separate processes open the same file for append, each process can write freely to the file without destroying the output being written by the other. The output from the two processes is intermixed in the order in which it is written to the file. Note that if the data is buffered, then it is not actually written until it is flushed.

If the **fopen** or **freopen** subroutine fails, a NULL pointer is returned.

Error Conditions
The **fopen** and **freopen** subroutines fail if one or more of the following are true:

EACCES Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by **mode** are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.

EINTR A signal was caught during the **fopen** function.

EISDIR The named file is a directory and **mode** requires write access.

EMFILE **FOPEN_MAX** file descriptors, directories and message catalogs are currently open in the calling process.

ENAMETOOLONG
The length of the **filename** string exceeds **PATH_MAX** or a pathname component is longer than **NAME_MAX**.

ENFILE The system file table is full.

ENOENT The named file does not exist or the **filename** argument points to an empty string.

ENOSPC The directory or file system that would contain the new file cannot be expanded; the file that was to be created does not exist.

ENOTDIR A component of the path prefix is not a directory.

ENXIO The named file is a character special or block special file, and the device associated with this special file does not exist.

EROFS The named file resides on a read-only file system and **mode** requires write access.

EINVAL The value of the **mode** argument is not valid.

ENOMEM Insufficient storage space is available.

ETXTBSY The file is a pure procedure (shared text) file that is being executed and **mode** requires write access.

The **fdopen** subroutine fails if one or more of the following is true:

EBADF The **fildes** argument is not a valid file descriptor.

EINVAL The **mode** argument is not a valid mode.

ENOMEM Insufficient space to allocate a buffer.

Related Information
In this book: "fclose, fflush" in topic 1.2.77, "fseek, rewind, ftell" in topic 1.2.86, "open, openx, creat" in topic 1.2.199, "setbuf, setvbuf" in topic 1.2.247, and " stdio" in topic 1.2.283.

1.2.83 fork, vfork

Purpose
Creates a new process.

Syntax

pid_t fork ( )

pid_t vfork ( )

Description
The **fork** system call creates a new process. The new process (child process) is an exact copy of the calling process (parent process). The created child process inherits the following attributes from the parent process:

Environmen
Close-on-exec flags (see "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71)
Signal handling settings (that is, **SIG_DFL**, **SIG_IGN**, **function address**)
Set-user-ID mode bi
Set-group-ID mode bi
Profiling on/off statu
Nice value (see "getpriority, setpriority, nice" in topic 1.2.111)
All attached shared libraries (see **shlib** command in *AIX Operating System Commands Reference*)
Process group I
Session I
TTY group ID (see "exit, _exit" in topic 1.2.73 and "sigaction, sigvec, signal" in topic 1.2.263)
Current director
Root director
File mode creation mask (see "umask" in topic 1.2.314)
System resource limits (see "ulimit" in topic 1.2.313)
Attached shared memory segments (see "shmat" in topic 1.2.258)
<LOCAL> alias pathname (see "getlocal, setlocal" in topic 1.2.102)
xvers string (see "getxvers, setxvers" in topic 1.2.129).

If the Transparent Computing Facility is installed, the following attributes are also inherited by the child process:

Execution site (see "rfork" in topic 1.2.237 to change execution sites)

Site path (see "getspath, setspath" in topic 1.2.122)

Execution site permissions (see "getxperm, setxperm" in topic 1.2.128)

setxuid bits (see "setxuid" in topic 1.2.256).

The child process differs from the parent process in the following ways:

The child process has a unique process ID. The child process ID als does not match any active process group ID.

The child process has as its parent process ID the process ID of th parent process.

The child process has its own copy of the parent's file descriptors However, each of the child process's file descriptors shares a common file pointer with the corresponding file descriptor of the parent process.

All **semadj** values are cleared. (For information about **semadj** values, see "semop" in topic 1.2.245.)

Process locks, text locks and data locks are not inherited by th child. (For information about locks, see "plock" in topic 1.2.205.)

The child process's trace flag (see the discussion of request 0 o "ptrace" in topic 1.2.212) is false regardless of the value of the parent process's trace flag.

The child process's **utime**, **stime**, **cutime**, and **cstime** are set to 0. (See "times" in topic 1.2.304.)

Any pending alarms are cleared in the child. (See "alarm" in topic 1.2.14.)

If the Transparent Computing Facility is installed, the **fork** system call only creates a new process on the local site (see "rfork" in topic 1.2.237).

Compatibility Note

The **vfork** system call is supported as a compatibility interface for older BSD programs, and can be used by compiling with Berkeley Compatibility Library (**libbsd.a**). Its function is superceded by **fork**. The 4.3BSD documentation warned that programs should not rely upon the unusual memory sharing semantics of **vfork**, since eventually "proper system sharing mechanisms" would be implemented. Accordingly, in AIX, **fork** and **vfork** have identical behavior.

Return Value

Upon successful completion, **fork** returns a value of 0 to the child process and returns the process ID of the child process to the parent process. If **fork** fails, a value of -1 is returned to the parent process, no child process is created, and **errno** is set to indicate the error.

Error Conditions

The **fork** system call fails if one or more of the following are true:

EAGAIN The system-imposed limit on the total number of processes executing would be exceeded.

EAGAIN The system-imposed limit on the total number of processes executing for a single user would be exceeded.

ENOMEM There is not enough space left for this process.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, "exit, _exit" in topic 1.2.73, "getpriority, setpriority, nice" in topic 1.2.111, "getspath, setspath" in topic 1.2.122, "getxvers, setxvers" in topic 1.2.129, "getxperm, setxperm" in topic 1.2.128, "plock" in topic 1.2.205, "ptrace" in topic 1.2.212, "rfork" in topic 1.2.237, "semop" in topic 1.2.245, "setxuid" in topic 1.2.256, "shmat" in topic 1.2.258, "sigaction, sigvec, signal" in topic 1.2.263, "times" in

topic 1.2.304, "ulimit" in topic 1.2.313, "umask" in topic 1.2.314, and "wait, waitpid" in topic 1.2.325.

The **shlib2** command in *AIX Operating System Commands Reference*.

Purpose
Sets and reads the 80387 control word.

Library
Math Library (**libm.a**)

Syntax

#include <sys/fpcontrol.h>

void fp_control(x)
unsigned int(x)

void fp_exmask(x)
unsigned int(x)

void fp_round(x)
unsigned int(x)

unsigned int fp_getcw()

unsigned int fp_getprecision()

void fp_restore()

void fp_exunmask(x)
unsigned int(x)

void fp_precision(x)
unsigned int(x)

unsigned int fp_getex()

unsigned int fp_getround()

unsigned int fp_getsw()

Description
The **fp_control** subroutine takes as an argument a new value for the 80387 control word.

The **fp_exmask** subroutine takes as an argument a bit mask corresponding to the bits in the 80387 control word interrupt mask fields. The bits are set in the control word causing the corresponding exceptions to be masked.

The **fp_exunmask** subroutine takes as an argument a bit mask corresponding to the bits in the 80387 control word interrupt mask fields. The bits are cleared in the control word causing the corresponding exceptions to be unmasked.

The **fp_round** subroutine takes as an argument the bit patterns corresponding to the bits in the 80387 control word rounding mode field. The rounding mode is set to the pattern specified.

The **fp_precision** subroutine takes as an argument the bit patterns corresponding to the bits in the 80387 control word precision mode field. The precision mode is set to the pattern specified.

The **fp_restore** subroutine restores the 80387 control word to the value and mode which existed before any of the above described subroutines were called. If none of the above described subroutines were called, then no action is taken.

The **fp_getcw** function returns the 80387 control word.

The **fp_getex** function returns the exception mask portion of the 80387

control word.

The **fp_getround** function returns the rounding mode portion of the 80387 control word.

The **fp_getprecision** function returns the precision mode portion of the 80387 control word.

The **fp_getsw** function returns the 80387 status word.

As an aid in specifying the exception mask bits and the rounding and precision mode bit patterns, a set of definitions has been provided in the include file **sys/fpcontrol.h**.

Exception masks:

FPM_INV_OP	0x0001
FPM_DENORM	0x0002
FPM_DIVIDE_0	0x0004
FPM_OVERFLOW	0x0008
FPM_UNDERFLOW	0x0010
FPM_PRECISION	0x0020

Rounding modes:

FPR_NEAR	0x0000
FPR_DOWN	0x0400
FPR_UP	0x0800
FPR_CHOP	0x0c00

Precision modes:

FPP_SINGLE	0x0000
FPP_DOUBLE	0x0200
FPP_EXTENDED	0x0300

These definitions can be used in a variety of ways. They can be ORed together to form an argument for the **fp_control** subroutine. If this is the case, any number of the exception mask symbols can be ORed with one of each of the rounding and precision mode values. For example:

```
fp_control(FPP_SINGLE | FPR_UP | FPM_PRECISION | FP_OVERFLOW);
```

selects single-precision results of calculations, rounding up to positive

infinity, and mask exception interrupts for precision and overflow exceptions.

The values can be used in the appropriate subroutine to modify a specific part of the 80387 control word. For example:

```
fp_precision(FPP_EXTENDED)
```

Note: Using a value with a subroutine which is not appropriate could produce an undesired result. The values can all be used with the **fp_control** subroutine as described above.

The values supplied in **fpcontrol.h** can be used with the **fp_getcw**, **fp_getround**, **fp_getex**, and **fp_getprecision** subroutines to determine what is currently selected. For example:

```
(fp_getex() & FPM_INV_OP)
```

returns a nonzero value if the invalid operation exception is masked.

```
(fp_getround() == FPR_CHOP)
```

returns a nonzero value if **chop** (truncate towards zero) rounding mode is currently selected in the 80387 control word.

1.2.84 fread, fwrite

Purpose
Performs binary input/output.

Library
Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>

int fread (ptr, size, nitems, stream)
void *ptr;
size_t size;
size_t nitems;
FILE *stream;

int fwrite (ptr, size, nitems, stream)
void *ptr;
size_t size;
size_t nitems;
FILE *stream;
```

Description
The **fread** subroutine copies **nitems** items of data from the input **stream** into an array beginning at the location pointed to by the **ptr** parameter. Each data item has the type ***ptr**.

The **fread** subroutine stops copying bytes if an end-of-file or error condition is encountered while reading from the input (1) specified by the **stream** parameter or (2) when the number of data items specified by the **nitems** parameter have been copied. **fread** leaves the file pointer of **stream**, if defined, pointing to the byte following the last byte read, if there is one. The **fread** subroutine does not change the contents of **stream**.

The **fwrite** subroutine appends **nitems** items of data of the type ***ptr** from the array pointed to by the **ptr** parameter to the output **stream**.

The **fwrite** subroutine stops writing bytes if (1) an error condition is encountered on **stream** or (2) when the number of items of data specified by the **nitems** parameter have been written. The **fwrite** subroutine does not change the contents of the array pointed to by the **ptr** parameter.

Return Value
The **fread** and **fwrite** subroutines return the number of items actually read or written. If the **nitems** parameter is negative or 0, no characters are read or written, and a value of 0 is returned.

Error Conditions
The **fread** subroutine fails if one or more of the following are true:

EAGAIN The **O_NONBLOCK** flag is set for the file descriptor underlying **stream** and the process would be delayed in the **fgetc** operation.

EBADF The file descriptor underlying **stream** is not a valid file descriptor open for reading.

EINTR The read operation was terminated due to the receipt of a signal

and no data was transferred.

EIO The process is a member of a background process attempting to read from its controlling terminal, the process is either ignoring or blocking the **SIGTTIN** signal or the process group is orphaned.

ENOMEM Insufficient storage space is available.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device.

The **fwrite** subroutine fails if one or more of the following is true:

EAGAIN The **O_NONBLOCK** flag is set for the file descriptor underlying **stream** and the process would be delayed in the write operation.

EBADF The file descriptor underlying **stream** is not a valid file descriptor open for writing.

EFBIG An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

EINTR The write operation was terminated due to the receipt of a signal and no data was transferred.

EIO The process is a member of a background process group attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking **SIGTTOU** and the process group of the process is orphaned.

ENOSPC There was no free space remaining on the device containing the file.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device.

EPIPE An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

ENOMEM Insufficient storage space is available.

Related Information

In this book: "fopen, freopen, fdopen" in topic 1.2.82, "getc, fgetc, getchar, getw, getwc, fgetwc, getwchar" in topic 1.2.91, "gets, fgets, getws, fgetws" in topic 1.2.117, "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf" in topic 1.2.208, "putc, putchar, fputc, putw, putwc, putwchar, fputwc" in topic 1.2.213, "puts, fputs, putws, fputws" in topic 1.2.216, "read, readv, readx" in topic 1.2.224, "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf, wsscanf" in topic 1.2.241, " stdio" in topic 1.2.283, and "write, writex" in topic 1.2.330.

1.2.85 frexp, ldexp, modf

Purpose
Manipulates parts of floating-point numbers.

Library
Standard C Library (**libc.a**)

Syntax

```
double frexp (value, eptr)  double ldexp (mant, exp)
double value;               double mant;
int *eptr;                  int exp;
                            double modf (value, iptr)
                            double value, *iptr;
```

Description
Every nonzero number can be written uniquely as **x** ¦ 2(**n**), where the mantissa (fraction), **x**, is in the range 0.5 = |**x**| < 1.0, and the exponent, **n**, is an integer. The internal representation of floating-point numbers uses this fact, storing a mantissa part and an exponent part.

The **frexp** subroutine returns the mantissa of the **value** parameter and stores the exponent in the location pointed to by the **eptr** parameter.

The **ldexp** subroutine returns the quantity **mant** ¦ 2(**exp**).

The **modf** subroutine returns the signed fractional part of the **value** parameter and stores the integral part in the location pointed to by the **iptr** parameter.

If the **ldexp** subroutine overflows, it returns HUGE and sets **errno** to ERANGE.

Error Conditions
The **modf** subroutine fails if the following is true:

EDOM The argument **value** is NaN.

The **ldexp** subroutine fails if the following is true:

ERANGE The value to be returned would have caused an overflow or underflow.

The **frexp** subroutine fails if the following is true:

EDOM The argument **value** is NaN or an infinity.

Related Information
In this book: "sputl, sgetl" in topic 1.2.280.

1.2.86 fseek, rewind, ftell

Purpose
Repositions the file pointer of a stream.

Library
Standard I/O Library (**libc.a**)

Syntax
#include <stdio.h>

```
int fseek (stream, offset, wvoiderewind (stream)
FILE *stream;               FILE *stream;
long offset;
int whence;                 long ftell (stream)
                            FILE *stream;
```

Description
The **fseek** subroutine sets the position of the next input or output operation on the I/O stream specified by the **stream** parameter. The position of the next operation is determined by the **offset** parameter, which can be either positive or negative.

The **fseek** subroutine sets the file pointer associated with the specified **stream** as follows:

If the **whence** parameter is 0, the pointer is set to the value of the **offset** parameter.

If the **whence** parameter is 1, the pointer is set to its current location plus the value of the **offset** parameter.

If the **whence** parameter is 2, the pointer is set to the size of the file plus the value of the **offset** parameter.

The **fseek** subroutine fails if attempted on a file that has not been opened using **fopen**. In particular, **fseek** cannot be used on a terminal, or on a file opened with **popen**.

Upon successful completion, **fseek** returns a value of 0. If **fseek** fails, a nonzero value is returned.

The **rewind** subroutine is equivalent to **fseek (stream, (long) 0, 0)**, except that it does not return a value.

The **fseek** and **rewind** subroutines undo any effects of the **ungetc** subroutine.

After an **fseek** or a **rewind**, the next operation on a file opened for update can be either input or output.

The **ftell** subroutine returns the offset of the current byte relative to the beginning of the file associated with the named **stream**.

Error Conditions
The **fseek** subroutine fails if the following is true:

EINVAL The resulting file-position indicator is set to a negative value.

The **fseek** and **ftell** subroutines fail if the following are true:

EBADF The **stream** had data to be flushed but the underlying file is not open for writing.

EFBIG An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.

EINVAL The **whence** argument is invalid.

ENOSPC There was no free space remaining on the device containing the file.

ESPIPE The file descriptor underlying **stream** is associated with a pipe or FIFO.

EAGAIN The **O_NONBLOCK** flag is set for the file descriptor and the process would be delayed in the write operation.

EINTR The write operation was terminated due to the receipt of a signal and no data was transferred.

EIO An I/O error occurred.

EPIPE An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

Related Information
In this book: "fopen, freopen, fdopen" in topic 1.2.82, "lseek" in topic 1.2.161, and " stdio" in topic 1.2.283.

1.2.87 fsync, fcommit

Purpose
Writes changes in a file to permanent storage.

Syntax

```
int fsync (fildes)
int fildes;

int fcommit (fildes)
int fildes;
```

Description
The **fsync** system call causes all modified data in the file open on **fildes** to be saved to permanent storage. The **fcommit** system call is a synonym for the **fsync** system call. Saving to permanent storage is sometimes called a ***commit operation***.

An **fsync** system call can be issued by a process executing at the node on which the file is stored or by a process executing at another node. In either case, the file is written to permanent storage at the node that holds the file.

If the file is open in **O_DEFERC** mode, **fsync** incorporates all data changes into a new version of the file. Old pages are freed and become unrecoverable. The file must be a regular file, not a pipe or special file, and the file descriptor must be open for writing. If the file has not been changed since it was open or last committed, this operation has no effect.

The file remains open after the **fsync** and a subsequent **fabort** system call on this file will roll back the file only to the latest commit checkpoint. **fsync** and **fabort** only affect the file's content and, except for the file's size, modification time and file status change time, the other file information is unaffected. Changes effected by system calls such as **chmod**, **chown**, and **utime** are atomic and are not aborted by **fabort**.

Changes made to a file opened with **O_DEFERC** are ordinarily made permanent when the file is closed or soon after, and thus, an explicit **fsync** is not usually necessary. If the file is open multiple times, the commit occurs on the final close of any write file descriptor. If this final close of the file is not done explicitly, but instead, the file is closed by **exit**, the file is still committed.

The **fsync** system call may be used to make a checkpoint of a file. This might be useful to protect against system failure which might cause file changes to be undone. Furthermore, if the file is replicated on several sites, **fsync** causes all copies of the file to be updated. For a file that continues to be open for a long time, checkpoints will tend to keep the copies of the file more closely up to date. This may be important if the network later becomes partitioned and some user or process is unable to access the latest version of the file. The user still may be able to access a more recent version of the file than could have been accessed in the absence of the checkpoint.

The **fsync** system call does not return until the file has been successfully updated at a site storing the file. If the file is replicated, the primary copy is written synchronously and other copies of the file are updated asynchronously as the other sites storing the file request the

changed file. A usage pattern with rapid, repetitive commits on a replicated file will probably be inefficient and should be avoided. Rapid commits may cause a slow or loaded site to skip a version of a file. If this happens, the secondary copy of the file is updated by propagating the entire file, rather than by propagating just the changed pages, as would be the case if the secondary copy was just one version out of date.

Warning: If the file is open both in **O_DEFERC** mode and the default open mode (**O_DEFERC** is not set in the **open** system call), by the same or different processes, changes may be written to permanent storage other than when an **fsync** system call is issued. Concurrent opens of this form should be avoided.

Return Value
Upon successful completion, **fsync** returns a value of 0. If **fsync** fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **fsync** system call fails if one or more of the following are true:

EIO I/O error.

EBADF **fildes** is not a valid file descriptor open for writing.

EINVAL **fildes** refers to a socket, not to a file.

ENOSPC There is no more space left on the file system.

If the Transparent Computing Facility is installed on your system, **fsync** can also fail if one or more of the following are true:

ESITEDN1 The storage site is down or has gone down since this file descriptor was issued.

ESITEDN2 The operation was terminated because a site failed.

Related Information
In this book: "close, closex" in topic 1.2.48, "fabort" in topic 1.2.75, "open, openx, creat" in topic 1.2.199, and "sync" in topic 1.2.295.

1.2.88 ftruncate, truncate

Purpose
Makes a file shorter.

Syntax

```
int ftruncate (fildes, lengtint truncate (path,length)
int fildes;                 char *path;
off_t length;               off_t length;
```

Description

The **ftruncate** and **truncate** system calls remove all data beyond **length** bytes from the beginning of the file. The **truncate** system call operates on the file specified in the **path** parameter. The **ftruncate** operates on the open file specified by the **fildes** parameter. Full blocks are returned to the file system so that they can be used again, and the file size is changed to the value of the **length** parameter. For **ftruncate**, however, if the file was opened with **O_DEFERC**, changes can be undone with **fabort**.

The **ftruncate** system call does not modify the seek pointer of the file.

If the **length** parameter is greater than the size of the file, the file size is not changed. Successful completion of the **ftruncate** or **truncate** system call clears the set-user-ID and set-group-ID attributes of the file.

Return Value
Upon successful completion, **ftruncate** and **truncate** return a value of 0. If **ftruncate** or **truncate** fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **ftruncate** and **truncate** system calls fail if one or more of the following are true:

EIO I/O error.

EINVAL The file is a directory, FIFO, or special file.

EAGAIN The write operation in **ftruncate** failed due to an enforced write lock on the file.

The **ftruncate** system call fails if the following are true:

EBADF **fildes** is not a valid file descriptor open for writing.

The **truncate** system call fails if the following are true:

ENOTDIR A component of the path prefix is not a directory.

ENOENT The named file does not exist.

EACCES A component of the path prefix denies search permission.

EACCES Write permission is denied for the named file.

EISDIR The named file is a directory.

EROFS The named file resides on a read-only file system.

ETXTBSY The file is a pure-procedure (shared text) file that is being executed.

EFAULT The **path** parameter points to a location outside of the process's allocated address space.

ENFILE The system file table or inode table is full.

ENAMETOOLONG
A component of the **path** parameter exceeded **NAME_MAX** characters or the entire **path** parameter exceeded **PATH_MAX** characters.

ENOENT A directory in the path prefix does not exist.

ENOENT A hidden directory was named, but no component inside it matched the process's current site path list.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

ELOOP A loop of symbolic links was detected.

If the Transparent Computing Facility is installed on your system, **ftruncate** can also fail if one or more of the following are true:

ESITEDN1 The storage site is down or in the case of **ftruncate**, has gone down since this file descriptor was issued.

ESITEDN2 The operation was terminated because a site failed.

If the Transparent Computing Facility is installed on your system, **truncate** can also fail if one or more of the following are true:

ENOSTORE **path** is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of **path** is replicated but not stored on any site which is currently up.

EROFS Write access is requested for a file on a replicated file system in which the primary copy is unavailable.

ENLDEV The named file is a non-**tty** character special file which corresponds to a device physically attached to another site in the cluster.

EINTR A signal was caught during the **truncate** system call.

Related Information
In this book: "fclear" in topic 1.2.76.

1.2.89 ftw

Purpose
Walks a file tree.

Library
Standard C Library (**libc.a**)

Syntax

#include <ftw.h>

```
int ftw (path, fn, depth)      int fn (filename, stat_ptr, file_t
char *path;                    char *filename;
int (*fn) ( );                 struct stat *stat_ptr;
int depth;                     int file_type;
```

Description
The **ftw** subroutine recursively searches the directory hierarchy that descends from the directory specified by the **path** parameter.

For each file in the hierarchy, the **ftw** subroutine calls the function specified by the **fn** parameter, passes it a pointer to a null-terminated character string containing the name of the file, a pointer to a **stat** structure containing information about the file, and an integer. (For information about the **stat** structure, see "stat.h" in topic 2.4.22.)

The *file_type* parameter to **fn** identifies the file type, and it has one of the following values:

FTW_F	Regular file
FTW_D	Directory
FTW_DNR	Directory that cannot be read
FTW_NS	A file for which **stat** could not be executed successfully.

If the integer is **FTW_DNR**, the files and subdirectories contained in that directory are not processed.

If the integer is **FTW_NS**, the **stat** structure contents are meaningless. An example of a file that causes **FTW_NS** to be passed to **fn** is a file in a directory for which you have read permission but not execute (search) permission.

The **ftw** subroutine finishes processing a directory before processing any of its files or subdirectories.

Symbolic links when encountered during the search process are processed according to the file type of the file to which they are symbolically linked. Processing of a symbolic link to a directory results in the invocation of **fn** on the symbolic link, but processing of the files or subdirectories within the symbolically linked directory is not done.

Hidden directories encountered during the search process are treated in the same manner as normal directories. Each hidden directory component will be processed in appropriate sequence.

The **ftw** subroutine continues the search until the directory hierarchy specified by the **path** parameter is completed, an invocation of the

function specified by the **fn** parameter returns a nonzero value, or an error is detected within **ftw**, such as an I/O error.

If the directory hierarchy is completed, the **ftw** subroutine returns a value of 0. If the function specified by the **fn** parameter returns a nonzero value, **ftw** stops its search and returns the value that was returned by the function. If the **ftw** subroutine detects an error, a value of -1 is returned and **errno** is set to indicate the error.

The **ftw** subroutine uses one file descriptor for each level in the tree. The **depth** parameter specifies the maximum number of file descriptors to be used. In general, the **ftw** subroutine runs faster if the value of the **depth** parameter is at least as large as the number of levels in the tree. However, the **depth** parameter must not be greater than the number of file descriptors currently available for use. If the value of the **depth** parameter is 0 or negative, the effect is the same as if it were 1.

Because the **ftw** subroutine is recursive, it is possible for it to terminate with a memory fault due to stack overflow when applied to very deep file structures.

The **ftw** subroutine uses the **malloc** subroutine to allocate dynamic storage during its operation. If **ftw** is terminated prior to its completion, such as by **longjmp** being executed by the function specified by the **fn** parameter or by an interrupt routine, then **ftw** cannot free that storage. The storage remains allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have the function specified by the **fn** parameter return a nonzero value the next time it is called.

Error Conditions
The **ftw** subroutine fails if the following are true:

EACCES Search permission is denied for any component of **path** or read permission is denied for **path**.

ENAMETOOLONG
The length of the **path** string exceeds **PATH_MAX**, or a pathname component is longer than **NAME_MAX**.

ENOENT The **path** argument points to the name of a file which does not exist or points to an empty string.

ENOTDIR A component of **path** is not a directory.

If **fn** points to a subroutine that encounters errors, additional **errno** values may be set.

Related Information
In this book: "malloc, free, realloc, calloc, valloc, alloca, mallopt, mallinfo" in topic 1.2.162, "setjmp, longjmp, _setjmp, _longjmp" in topic 1.2.250, "sigaction, sigvec, signal" in topic 1.2.263, and "statx, fstatx, stat, fstat, fullstat, ffullstat, lstat" in topic 1.2.282.

1.2.90 gamma, lgamma

Purpose
Computes the logarithm of the gamma function.

Library
Math Library (**libm.a**)

Syntax

```
#include <math.h>

extern int signgam;

double gamma (x)
double x;

double lgamma (x)
double x;
```

Description
The **gamma** subroutine returns **ln**(|&Gamma.(**x**)|) and **lgamma** returns **ln**&Gamma.(**x**), where &Gamma.(**x**) is defined as:

The sign of &Gamma.(**x**) is stored in the external integer variable **signgam**. The **x** parameter cannot be a nonpositive integer.

If the **x** parameter is a nonpositive integer, **gamma** and **lgamma** return HUGE, sets **errno** to EDOM, and writes a DOMAIN error message to standard error.

If the correct value overflows, **gamma** and **lgamma** return HUGE and sets **errno** to ERANGE.

You can change the error handling procedures with the **matherr** subroutine.

Examples
The following C program fragment calculates &Gamma.(**x**) and stores the result in **y**:

```
errno = 0;
y = gamma(x);
if (errno == 0)
   y = signgam * exp(y);
else
   perror("Error in gamma function");
```

Error Conditions
The **gamma** and **lgamma** subroutines fail if one or more of the following is true:

EDOM The value of **x** is a non-positive integer or NaN.

ERANGE The value to be returned would have caused overflow.

Related Information
In this book: "matherr" in topic 1.2.163 and "cbrt, exp, expm1, log, log10, log1p, pow, sqrt" in topic 1.2.28.

1.2.91 getc, fgetc, getchar, getw, getwc, fgetwc, getwchar

Purpose
Gets a character, a wide character, or word from an input stream.

Library
Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>

int getc (stream)
FILE *stream;

int fgetc (stream)
FILE *stream;

wchar_t getwc (stream)
FILE *stream;

wchar_t fgetwc (stream)
FILE *stream;

int getchar ( )

int getw (stream)
FILE *stream;

wchar_t getwchar ( )
```

Description
The **getc** macro returns the next character (byte) from the input specified by the **stream** parameter and moves the file pointer, if defined, ahead one character in **stream**. **getc** is a macro and cannot be used where a subroutine is necessary; for example, a subroutine pointer cannot point to it.

Because it is implemented as a macro, **getc** does not work correctly with a **stream** parameter that has side effects. In particular, the following does not work:

```
getc(*f++)
```

In cases like this, use the **fgetc** subroutine instead.

The **fgetc** subroutine performs the same function as **getc**, but **fgetc** is a genuine subroutine, not a macro. The **fgetc** subroutine runs more slowly than **getc**, but takes less space.

The **getchar** macro returns the next character from the standard input stream, **stdin**. Note that **getchar** is also a macro.

The **getw** subroutine returns the next word (**int**) from the input specified by the **stream** parameter and increments the associated file pointer, if defined, to point to the next word. The size of a word varies from one machine architecture to another. The **getw** subroutine returns the constant EOF at end-of-file or when an error occurs. Since EOF is a valid integer value, **feof** and **ferror** should be used to check the success of **getw**. The **getw** subroutine assumes no special alignment in the file.

Because of possible differences in word length and byte ordering from one machine architecture to another, files written using **putw** are machine-dependent and may not be readable using **getw** on a different type of processor.

The **getwchar** subroutine is equivalent to **getwc** with the argument **stdin**.

The **fgetwc** subroutine is equivalent to **fgetc** except a wide character is returned.

The **getwc** function obtains the next wide character, if present, from the input stream pointed to by **stream**, and advances the associated file position indicator for the stream, if defined. The file position indicator is advanced for each multibyte character obtained.

The **fgetwc** and the **getwchar** subroutines return the next wide character which corresponds to a multibyte character from the input stream pointed to by **stream**. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **fgetwc** returns WEOF. If a read error occurs, the error indicator for the stream is set and **fgetwc** returns WEOF.

Return Value
The **getc**, **fgetc**, **getchar**, and **getw** return the integer constant EOF at end-of-file or error, while the **getwc**, **fgetwc**, and **getwchar** return WEOF at end-of-file or error.

Error Conditions
These subroutines fail if one or more of the following are true:

EAGAIN The **O_NONBLOCK** flag is set for the file descriptor underlying **stream** and the process is delayed in the **fgetc()** operation.

EBADF The file descriptor underlying **stream** is not a valid file descriptor open for reading.

Note: If a wide character routine fails and **errno** is not set, this indicates that the translation from file code to wide code has failed.

EINTR The read operation was terminated due to the receipt of a signal and no data was transferred.

EIO The process is a member of a background process attempting to read from its controlling terminal, the process is either ignoring or blocking the **SIGTTIN** signal or the process group is orphaned.

ENOMEM Insufficient storage space is available.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device.

Related Information
In this book: "feof, ferror, clearerr, fileno" in topic 1.2.79, "fopen, freopen, fdopen" in topic 1.2.82, "fread, fwrite" in topic 1.2.84, "gets, fgets, getws, fgetws" in topic 1.2.117, "NLgetctab" in topic 1.2.190, "putc, putchar, fputc, putw, putwc, putwchar, fputwc" in topic 1.2.213, "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf, wsscanf" in topic 1.2.241, and " stdio" in topic 1.2.283.

1.2.92 getcwd

Purpose
Gets the path name of the current directory.

Library
Standard C Library (**libc.a**)

Syntax

```
char *getcwd (buf, size)
char *buf;
int size;
```

Description
The **getcwd** subroutine returns a pointer to a string containing the path name of the current directory. The value of the **size** parameter must be at least two greater than the length of the path name to be returned.

If the **buf** parameter is a NULL pointer, the **getcwd** subroutine will, using the **malloc** subroutine, obtain the number of bytes of free space as specified by the **size** parameter. In this case, the pointer returned by the **getcwd** subroutine can be used as the parameter in a subsequent call to **free**.

If the **getcwd** subroutine fails, NULL is returned and **errno** is set to indicate the error. The **getcwd** subroutine fails if the **size** parameter is not large enough or if an error occurs in a lower-level function.

Error Conditions
If any of the following conditions occur, the **getcwd** function returns a value of NULL and sets **errno** to the corresponding value:

EINVAL The **size** argument is less than or equal to 0.

ERANGE The **size** argument is greater than 0 but smaller than the length of the path name plus 1.

EACCES Read or search permission was denied for a component of **pathname**.

ENOMEM Insufficient storage space is available.

Related Information
In this book: "malloc, free, realloc, calloc, valloc, alloca, mallopt, mallinfo" in topic 1.2.162 and " popen, pclose, rpopen" in topic 1.2.207.

The **pwd** command in *AIX Operating System Commands Reference*.

1.2.93 getdtablesize

Purpose
Gets descriptor table size.

Syntax

int getdtablesize ( )

Description
The **getdtablesize** system call returns the size of a process descriptor table, which has at least 20 slots for each process. Table entries consist of small integers starting at 0.

In AIX, **getdtablesize** returns NOFILE, which is 200.

Related Information
In this book: "close, closex" in topic 1.2.48, "dup" in topic 1.2.64, "dup2" in topic 1.2.65, "open, openx, creat" in topic 1.2.199, and "select" in topic 1.2.242.

1.2.94 getenv, NLgetenv

Purpose
Returns the value of an environment variable.

Library
Standard C Library (**libc.a**)

Syntax

```
char *getenv (name)          char *NLgetenv (name)
char *name;                  char *name;
```

Description

The **getenv** subroutine searches the environment list for a string of the form **name=value**. Environment variables are sometimes called shell variables since they are frequently set with shell commands.

Note: **NLgetenv** is a front-end to the **nl_langinfo** subroutine, and the values returned by **nl_langinfo** are initialized by calling **setlocale** for the current locale (see "setlocale" in topic 1.2.251 and "nl_langinfo" in topic 1.2.198). If **nl_langinfo** returns a NULL, then **NLgetenv** calls **getenv**.

Return Value
The **getenv** subroutine returns a pointer to the **value** in the current environment if such a string is present. If such a string is not present, a NULL pointer is returned.

Related Information
In this book: "NLgetfile" in topic 1.2.191, "putenv" in topic 1.2.214, "nl_langinfo" in topic 1.2.198, "setlocale" in topic 1.2.251, and "environment" in topic 2.4.6.

The **sh** command in *AIX Operating System Commands Reference*.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.95 getfsent, getfsspec, getfsfile, getfstype, setfsent, endfsent

Purpose
Gets information about a file system.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <fstab.h>


struct fstab *getfsent ( )  struct fstab *getfstype (type)
                            char *type;


struct fstab *getfsspec  (spint setfsent ( )
char *spec;


struct fstab *getfsfile (filint endfsent ( )
char *file
```

Description
The **getfsent, getfsspec, getfstype**, and **getfsfile** subroutines each return a pointer to a structure which contains information about a file system. The header file **fstab.h** describes the structure.

These routines are provided for 4.3BSD compatibility. Since AIX uses the file **/etc/filesystems** rather than **/etc/fstab**, these routines expect a file as described in "filesystems" in topic 2.3.18. They return a pointer to an **fstab** structure in the static area, which must be copied if it is to be saved.

The **getfsent** subroutine reads the next record of the file, opening the file if necessary.

The **setfsent** subroutine opens the file and positions to the first record.

The **endfsent** subroutine closes a file.

The **getfsspec** and **getfsfile** subroutines sequentially search from the beginning of the file until a matching special file name or file system file name is found, or until the end of the file is encountered. The **getfstype** subroutine does likewise, matching on the file system type field.

Return Value
If an end-of-file condition or an error is encountered on reading, subroutines **getfsent**, **getfsspec**, **getfsfile**, and **getfstype** return a NULL pointer. The subroutines **setfsent** and **endfsent** return 1 on success and 0 on error.

Related Information

"Introduction to International Character Support" in *Managing the AIX Operating System*.

1.2.96 getgrent, getgrgid, getgrnam, setgrent, endgrent

Purpose
Accesses group file entries.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <grp.h>

struct group *getgrent ( )      struct group *getgrnam (name)
                                char *name;
struct group *getgrgid (gid)
int gid;                        void setgrent ( )
                                void endgrent ( )
```

Description
The **getgrent**, **getgrgid**, and **getgrnam** subroutines return a pointer to a structure containing the broken-out fields of a line in the **/etc/group** file. The **group** structure is defined in the **grp.h** header file, and it contains the following members:

```
char  *gr_name;    /* The name of the group */
char  *gr_passwd;  /* The encrypted group password */
gid_t gr_gid;      /* The numerical group ID */
char  **gr_mem;    /* Array of pointers to member names */
```

The **getgrent** subroutine, when first called, returns a pointer to the first group structure in the file. On the next call, it returns a pointer to the next group structure in the file. You can call **getgrent** repeatedly to search the entire file.

The **getgrgid** subroutine searches from the beginning of the file until it finds a numerical group ID matching the **gid** parameter. The subroutine then returns a pointer to the structure in which it was found.

The **getgrnam** subroutine searches from the beginning of the file until it finds a group name matching the **name** parameter. The subroutine then returns a pointer to the structure in which it was found.

These subroutines return a pointer to a **group** structure contained in the static area, which must be copied if it is to be saved. If an end-of-file condition or an error is encountered on reading, these functions return a NULL pointer.

The **setgrent** subroutine rewinds the group file to allow repeated searches.

The **endgrent** subroutine closes the group file when processing is complete.

File
/etc/group

Error Conditions
The **getgrgid** and **getgrnam** subroutines fail if one or more of the following are true:

EIO An I/O error has occurred.

EINTR A signal was caught during the function.

EMFILE Too many file descriptors are currently open for the process.

ENFILE The system file table is full.

Related Information
In this book: "getlogin" in topic 1.2.103, "getpwent, getpwuid, getpwnam, setpwent, endpwent" in topic 1.2.114, and "group" in topic 2.3.26.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

1.2.97 getgroups

Purpose
Gets the group access list.

Syntax

```
#include <grp.h>

int getgroups (ngroups, gidset)
int ngroups, *gidset;
```

Description
The **getgroups** system call gets the current group access list of the user process. The list is stored in the array pointed to by the **gidset** parameter. The **ngroups** parameter indicates the number of entries that can be stored in this array. **getgroups** never returns more than NGROUPS entries. (NGROUPS is a constant defined in the **grp.h** header file.)

As a special case, if the **ngroups** parameter is 0, the number of entries in the group access list of the user process is returned. The array pointed to by the **gidset** parameter is not modified.

Return Value
Upon successful completion, the **getgroups** system call returns the number of elements stored into the array pointed to by the **gidset** parameter. If **getgroups** fails, then a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **getgroups** system call fails if the following is true:

EFAULT The **ngroups** and **gidset** parameters specify an array that is partially or completely outside of the process's allocated address space.

EINVAL The argument, **ngroups**, is smaller than the number of entries in the current group access list.

Related Information
In this book: "initgroups" in topic 1.2.135 and "setgroups" in topic 1.2.249.

1.2.98 gethostbyaddr, gethostbyname, sethostent, endhostent

Purpose
Get network host entry.

Library
Internet Library (**libc.a**)

Syntax
```
#include <netdb.h>

                struct hostent *gethostbyaddr(advoidlsethostent(stayopen)
                char *addr;                     int stayopen;
                int len, type;
                                                void endhostent()
                struct hostent *gethostbyname(name)
                char *name;
```

Description
The **gethostbyname** and **gethostbyaddr** subroutines each return a pointer to an object. This object is a **hostent** structure, which contains information obtained from the name server program, or a field from a line in the **/etc/hosts** file (the network host data base). If the local name server is not running, these routines do a lookup in **/etc/hosts**.

The **hostent** structure is defined in the **netdb.h** header file, and it contains the following members:

```
          char    *h_name;        /* official name of host */
          char    **h_aliases;    /* alias list */
          int     h_addrtype;     /* host address type */
          int     h_length;       /* length of address */
          char    **h_addr_list;  /* list of addresses from name server */

  #define h_addr  h_addr_list[0]  /* address, for backward compatibility */
```

The members of this structure are:

h_name Official name of the host.

h_aliases A zero terminated array of alternate names for the host.

h_addrtype
The type of address being returned. The subroutine always sets this value to **AF_INET**.

h_length The length, in bytes, of the address.

h_addr_list
An array, terminated by 0, of pointers to the network addresses for the host. Host addresses are returned in network byte order.

h_addr The first address in **h_addr_list**, provided for backward compatibility.

The **sethostent** subroutine allows a request for the use of a connected socket using TCP for queries. If the **stayopen** parameter is nonzero, an option is set to send all queries to the name server using TCP and to retain the connection after each call to **gethostbyname** or **gethostbyaddr**.

The **endhostent** subroutine closes the TCP connection.

The **gethostbyname** and **gethostbyaddr** subroutines query the name server or search the file sequentially from its beginning until finding a matching host name or host address, or until encountering the end of the file. Host addresses are supplied in network order.

Return Value
The **gethostbyname** and **gethostbyaddr** subroutines return a pointer to a **hostent** structure on success.

Note: The return value points to static data that is overwritten by subsequent calls.

A NULL pointer (0) is returned if an error occurs or the end of the file is reached and the **h_errno** variable is set to indicate the error.

Error Conditions
The **gethostbyname** and **gethostbyaddr** subroutines fail if one or more of the following are true:

HOST_NOT_FOUND
The host specified by the **name** parameter was not found.

TRY_AGAIN
The local server did not receive a response from an authoritative server. Try again later.

NO_RECOVERY
This error code indicates an unrecoverable error.

NO_ADDRESS
The requested **name** is valid but does not have an Internet address at the name server.

Files

/etc/hosts Host name data base.

/etc/resolv.conf Name server and domain name data base.

Related Information
In this book: "Related Network Publications" in topic 1.2.277.4.

The discussion of **host**, **named**, and **resolv.conf** in *AIX TCP/IP User's Guide*.

1.2.99 gethostid, sethostid

Purpose
Gets or sets the unique identifier of the current Internet host.

Syntax

```
int gethostid ( )          int sethostid (hostid)
                           int hostid;
```

Description

The **gethostid** system call returns the 32-bit identifier for the current host, as set by **sethostid**.

The **sethostid** system call establishes a 32-bit identifier for the current host that is intended to be unique. Often, this is a DARPA Internet address for the local machine.

This system call can only be used by processes with an effective user ID of superuser.

Return Value

Upon successful completion, the **gethostid** system call returns the identifier for the current host, and the **sethostid** system call returns a value of 0. If the **gethostid** or **sethostid** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **gethostid** or **sethostid** system call fails if the following is true:

EINVAL There are no IP interfaces available. IBM AIX TCP/IP is not installed on this system.

The **sethostid** system call also fails if the following is true:

EPERM The calling process did not have an effective user ID of superuser.

Related Information
In this book: "getsockname" in topic 1.2.120.

The **hostname** command in *AIX TCP/IP User's Guide*.

1.2.100 gethostname, sethostname

Purpose
Gets or sets the name of the current host.

Syntax

```
int gethostname (name, namelint sethostname (name, namelen)
char *name;                 char *name;
int namelen;                int namelen;
```

Description

The **gethostname** system call returns the standard host name of the current host, as set by **sethostname**. The parameter **namelen** specifies the size of the **name** array. The returned name is null-terminated unless insufficient space is provided.

The **sethostname** system call sets the name of the host machine **name** with the length **namelen**. This system call can only be used by processes with an effective user ID of superuser. In the AIX Operating System, the host name of a machine is usually set by AIX TCP/IP in its initialization program (**/etc/rc.tcpip**).

Return Value

Upon successful completion, a value of 0 is returned. If the **gethostname** or **sethostname** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The system call fails if one or more of the following are true:

EFAULT The **name** parameter or **namelen** parameter gives an address that is not valid.

EPERM The calling process did not have an effective user ID of superuser.

Related Information
In this book: "gethostid, sethostid" in topic 1.2.99.

The **hostname** command in *AIX TCP/IP User's Guide*.

1.2.101 getitimer, setitimer

Purpose
Gets and sets value of internal timer.

Syntax

```
#include <sys/time.h>

int getitimer (which, value)int setitimer (which, value, ovalu
int which;                  int which;
struct itimerval *value;    struct itimerval *value, *ovalue;
```

Description
The **getitimer** system call returns the current value of the timer specified in **which**. The **setitimer** system call sets the timer in **which** to the specified **value**, returning the previous value of the timer if **ovalue** is not 0.

The **itimerval** structure describes the timer value, as defined in the **sys/time.h** header file, and it contains the following members:

```
struct timeval  it_interval;    /* timer interval */
struct timeval  it_value;       /* current value  */
```

Setting **it_interval** to 0 disables the timer after it expires. An **it_interval** other than 0 specifies a value used to reload **it_value** when the timer expires.

The **it_value** disables the timer immediately when set to 0. An **it_value** other than 0 indicates the time of the next timer expiration.

Time values smaller than the resolution of the system clock are rounded up to its resolution, defined by IHZ, which is included in **sys/param.h**.

The **which** parameter is set to one of the following:

ITIMER_REAL The timer decrements in real time. When it expires, the system delivers a **SIGALRM** signal.

ITIMER_VIRTUAL The time decrements in process virtual time; it runs only when the process is executing. When it expires, the system delivers a **SIGVTALRM** signal.

ITIMER_PROF The timer decrements both in process virtual time and when the operating system is executing on behalf of the process. When it expires, the system delivers a **SIGPROF** signal.

Return Value
Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **getitimer** or **setitimer** system call fails if one or more of the following is true:

EFAULT The **value** or **ovalue** parameter points to a location outside of the process's allocated address space.

EINVAL The **value** parameter specifies a time too large to handle.

EINVAL The **which** parameter specifies an illegal value.

Related Information
In this book: "gettimeofday, settimeofday, ftime" in topic 1.2.123 and "sigaction, sigvec, signal" in topic 1.2.263.

1.2.102 getlocal, setlocal

Purpose
Manages the **<LOCAL>** alias.

Syntax

```
int getlocal(localname, maxlength)
char *localname;
int maxlength;

int setlocal(localname)
char *localname;
```

Description
The **getlocal** system call returns the calling process's alias for **<LOCAL>**. The alias path name is returned in the **localname** buffer. The **setlocal** system call sets the value of the current **<LOCAL>** alias.

The **<LOCAL>** alias is evaluated whenever the system encounters a symbolic link beginning with the string **<LOCAL>**. At that point, it substitutes the alias path name for **<LOCAL>** in the path name and continues path name interpretation normally. If the alias has a leading '/', then the name evaluation starts from the root directory, otherwise the directory containing the symbolic link is used to further evaluate the new name (identical to the semantics of ordinary symbolic links).

Return Value
Upon successful completion, a value of 0 is returned to the calling process. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Results
The **getlocal** system call fails if any of the following are true:

EINVAL The alias name is longer than **maxlength** (the length of the **localname** buffer).

EFAULT **localname** is not a region inside the user's address space.

The **setlocal** system call fails if any of the following are true:

ENOTDIR A component of the path is not a directory.

ENOENT The path name given does not exist.

ENOENT A null path name was provided.

ENOENT A hidden directory was named, but no component inside it matched the process's current site path list.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

EACCES Search permission is denied on a component of the path prefix.

EFAULT **path** points outside the process's allocated address space.

EFAULT The name pointed at by **localname** is too long (current limit is 30 characters).

If the Transparent Computing Facility is installed on your system, **getlocal** can also fail if one or more of the following are true:

ESITEDN1 **path** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE **path** is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of **path** is replicated but not stored on any site which is currently up.

Related Information
In this book: "Creation and Execution" in topic 1.1.4.3.1 and "symlink" in topic 1.2.294.

1.2.103 getlogin

Purpose
Gets the user's login name.

Library
Standard C Library (**libc.a**)

Syntax

char *getlogin ( )

Description
The **getlogin** subroutine returns a pointer to the login name as found in the process's user info area as returned by **getuinfo**. Use the **getlogin** subroutine in conjunction with the **getpwnam** subroutine to locate the correct password file entry when the same user ID is shared by several login names.

If the **getlogin** subroutine is called within a process that is not attached to a terminal, it returns a NULL pointer. The correct procedure for determining the login name is to call **cuserid**, or to call **getlogin** and if it fails, then to call **getpwuid**.

If the login name is not found, **getlogin** returns a NULL pointer.

Warning: The **getlogin** subroutine returns a pointer to a static area that is overwritten by successive calls.

Error Conditions
The **getlogin** subroutine fails if one or more of the following are true:

EMFILE Too many file descriptors are in use by this process.

ENFILE The system file table is full.

ENXIO The calling process has no controlling terminal.

Related Information
In this book: "cuserid" in topic 1.2.57, "getgrent, getgrgid, getgrnam, setgrent, endgrent" in topic 1.2.96, "getpwent, getpwuid, getpwnam, setpwent, endpwent" in topic 1.2.114, and "getuinfo" in topic 1.2.125.

1.2.104 getmntent, setmntent, addmntent, endmntent, hasmntopt

Purpose
Gets file system descriptor file entry.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <stdio.h>
#include <mntent.h>


              struct mntent *getmntent(filint addmntent(filep,mnt)
              FILE *filep;                FILE *filep
                                          struct mntent *mnt;


              FILE *setmntent(filep, type)int endmntent(filep)
              char *filep;                FILE *filep;
              char *type;


              char *hasmntopt(mnt, opt)
              struct mntent *mnt;
              char *opt;
```

Description
These routines are used to access the mounted file system description file **/etc/mtab**. In a TCF cluster, **/etc/mtab** is a symbolic link to a local-only file. To find out about file systems mounted on other sites, you must specify that site's **<LOCAL>/mtab** file to **setmntent**.

The **setmntent** routine opens a mounted file system description file and returns a file pointer which can then be used with **getmntent**, **addmntent**, or **endmntent**. The **type** argument is the same as in **fopen**. The **getmntent** routine reads the next line from **filep** and returns a pointer to an object with the following structure containing the broken-out fields of a line in the mounted file system description file, **<mntent.h>**.

The fields have meanings described in **mntent**:

```
struct mntent {
       char     *mnt_fsname;   /* file system name */
       char     *mnt_dir;      /* file system path prefix */
       char     *mnt_type;     /* ufs, nfs, swap, or xx */
       char     *mnt_opts;     /* ro, quota, etc. */
       int      mnt_freq;      /* dump frequency, in days */
       int      mnt_checkno;   /* check number for parallel fsck */
       char     mnt_flags;     /* file system flags */
       gfs_t    mnt_gfs;       /* global file system number */
       pckno_t  mnt_pack;      /* pack number */
       long     mnt_time;      /* time when mounted */
};
```

The **addmntent** routine adds the **mntent** structure **mnt** to the end of the open file **filep**. Note that **filep** has to be opened for writing if this is to work. The **hasmntopt** routine scans the **mnt_opts** field of the **mntent** structure **mnt** for a substring that matches **opt**. It returns the address of

the substring if a match is found; otherwise, it returns the value 0. The **endmntent** routine closes the file.

Warning: The returned **mntent** structure points to static information that is overwritten in each call.

Return Value

A NULL pointer (0) is returned if an error occurs or the end of a file is reached.

File

/etc/mtab

Related Information

In this book: "mntent, mtab" in topic 2.3.40.

1.2.105 getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent

Purpose
Gets network entry.

Library
Internet Library (**libc.a**)

Syntax
#include <netdb.h>

```
struct netent *getnetent ( )void setnetent (stayopen)
                                int stayopen;
struct netent *getnetbyname (name)
char *name;                     void endnetent ( )
struct netent *getnetbyaddr (net, type)
long net;
int type;
```

Description
The **getnetent**, **getnetbyname**, and **getnetbyaddr** subroutines each return a pointer to an object. This object is a **netent** structure, which contains the field of a line in the **/etc/networks** file (the network data base). The **netent** structure is defined in the **netdb.h** header file, and it contains the following members:

```
char            *n_name;              /* official name of net */
char            **n_aliases;          /* alias list           */
int             n_addrtype;           /* net number type      */
unsigned long   n_net;                /* net number           */
```

The members of the structure are defined below:

n_name Official name of the network.

n_aliases An array, terminated with a 0, of alternate names for the network.

n_addrtype The type of network number being returned. **AF_INET** (defined in **<sys/socket.h>**) is the only valid value for this field.

n_net The network number. Network numbers are returned in machine byte order.

The **getnetent** subroutine reads the next line of the file. If the file is not open, **getnetent** opens it.

The **setnetent** subroutine opens and rewinds the file. If the **stayopen** parameter is 0, the net data base is closed after each call to **getnetbyname** or **getnetbyaddr**. Otherwise, the file is not closed after each call.

The **endnetent** subroutine closes the file.

The **net** parameter of **getnetbyaddr** subroutine contains the number of the network to be located. The **type** parameter specifies the address family for the network. The only supported value is **AF_INET**.

The **getnetbyname** and **getnetbyaddr** subroutines search the file sequentially from its beginning until:

Finding a matching net name, for **getnetbyname**

Finding a matching net number and type for **getnetbyaddr**

Encountering the end of the file, for either routine

Network numbers are supplied in host order.

Return Value

A pointer to a **netent** structure is returned on success.

Note: The return value points to static data that is overwritten by subsequent calls.

A NULL pointer (0) is returned if an error occurs or the end of the file is reached.

File

/etc/networks Network name data base.

Related Information

The discussion of **/etc/networks** in *AIX TCP/IP User's Guide*.

1.2.106 getopt

Purpose
Gets flag letters from the argument vector.

Library
Standard C Library (**libc.a**)

Syntax

```
int getopt (argc, argv, optsextern char *optarg;
int argc;
char **argv;                extern int optind, opterr;
char *optstring;
```

Description
The **getopt** subroutine returns the next flag letter in the **argv** parameter list that matches a letter in the **optstring** parameter. The **getopt** subroutine is an aid to help programs interpret shell command-line flags that are passed to them.

The **optstring** parameter is a string of recognized flag letters. If a letter is followed by a colon, the flag is expected to take a parameter that may or may not be separated from it by white space. The **optarg** external variable is set to point to the start of the flag's parameter on return from the **getopt** subroutine.

The **getopt** subroutine places the **argv** index of the next argument to be processed in **optind**. **optind** is externally initialized to 1 so that **argv**[0] is not processed.

When all flags have been processed (that is, up to the first nonflag argument), the **getopt** subroutine returns EOF. The special flag **--** (dash dash) can be used to delimit the end of the flags; EOF is returned, and **--** is skipped.

The **getopt** subroutine prints an error message on **stderr** and returns **(int) '?'** (question mark) when it encounters a flag letter that is not included in the **optstring** parameter. You can disable this error message by setting **opterr** to 0.

Examples
The following code fragment processes the flags for a command that can take the mutually exclusive flags **a** and **b**, and the flags **f** and **o**, both of which require parameters.

```
#include <unistd.h>      /* Needed for access system call constants */
#include <stdio.h>

main (argc, argv)
int argc;
char **argv;
{
   int c;
   int aflg=0;
   int bflg=0;
   extern int optind;
   extern char *optarg;
   .
   .
```

```
      .
      while ((c = getopt(argc, argv, "abf:o:")) != EOF)
      {
         switch (c)
         {
            case 'a':
               if (bflg)
                  errflg++;
               else
                  aflg++;
               break;
            case 'b':
               if (aflg)
                  errflg++;
               else
                  bflg++;
               break;
            case 'f':
               ifile = optarg;
               break;
            case 'o':
               ofile = optarg;
               break;
            case '?':
               errflg++;
         } /* case */

         if (errflg)
         {
            fprintf(stderr, "usage:...");
            exit(2);
         }
      } /* while */
         if (aflg) printf ("-a flag seen\n");
         if (bflg) printf ("-b flag seen\n");

      for ( ; optind < argc; optind++)
      {
         if (access(argv[optind], R_OK))
         {
         .
         .
         .
         }
      } /* for  */
   }    /* main */
```

Related Information

The **getopt** command in *AIX Operating System Commands Reference*.

1.2.107 getpagesize

Purpose
Gets system page size.

Syntax

int getpagesize ()

Description
The **getpagesize** system call returns the number of bytes in a page. Page granularity is the granularity of many of the memory management calls.

The page size is a *system* page size and may not be the same as the underlying hardware page size.

Related Information
In this book: "brk, sbrk" in topic 1.2.21 and "getrlimit, setrlimit, vlimit" in topic 1.2.115.

1.2.108 getpass

Purpose
Reads a password.

Library
Standard C Library (**libc.a**)

Syntax

```
char *getpass (prompt)
char *prompt;
```

Description
The **getpass** subroutine writes the **prompt** string to standard error output, disables echoing, and reads up to a new-line character or EOF from the file **/dev/tty**.

It returns a pointer to a null-terminated string of no more than eight characters. This return value points to data that is overwritten by successive calls. If the **/dev/tty** file cannot be opened, a NULL pointer is returned.

An interrupt terminates input and sends an interrupt signal to the calling program before returning.

File
/dev/tty

Error Conditions
The **getpass** subroutine fails if one or more of the following are true:

EINTR The **getpass** subroutine was interrupted by a signal.

ENXIO The process does not have a controlling terminal.

Related Information
In this book: "crypt, encrypt, setkey" in topic 1.2.52.

1.2.109 getpeername

Purpose
Gets the name of the connected peer.

Syntax

```
int getpeername (s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

Description

The **getpeername** system call returns the name of the ***peer***, or connected socket, that is connected to the socket specified by the **s** parameter. You should initialize the **namelen** to indicate the amount of space pointed to by **name**. On return, it contains the actual size of the name returned (in bytes).

Return Value

Upon successful completion, a value of 0 is returned. If the **getpeername** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The system call fails if one or more of the following are true:

EBADF The **s** parameter is not valid.

ENOTSOCK The **s** parameter refers to a file, not a socket.

ENOTCONN The socket is not connected.

ENOBUFS Insufficient resources were available in the system to complete the call.

EFAULT The **addr** parameter is not in a writable part of the user address space.

Related Information
In this book: "bind" in topic 1.2.20, "getsockname" in topic 1.2.120, and "socket" in topic 1.2.275.

1.2.110 getpid, getpgrp, getppid

Purpose
Gets the process, process group, and parent process IDs.

Syntax

pid_t getpid ( )

pid_t getpgrp ( )

pid_t getppid ( )

Description
The **getpid** system call returns the process ID of the calling process.

The **getpgrp** system call returns the process group ID of the calling process.

The **getppid** system call returns the process ID of the calling process's parent process.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes become "orphans" and the parent process ID of each remaining child process is set to -1. However, in this case, the **getppid** system call returns 1 to allow for POSIX conformance.

Related Information
In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, " fork, vfork" in topic 1.2.83, "setpgid, setpgrp, setsid" in topic 1.2.252, and "sigaction, sigvec, signal" in topic 1.2.263.

1.2.111 getpriority, setpriority, nice

Purpose
Gets or sets program scheduling priority.

Library
Standard C Library (**libc.a**)
BSD Compatibility Library (**libbsd.a**)

Syntax

```
#include <sys/resource.h>


        prio= getpriority(which, whoint nice(incr)
        int prio, which, who;       int incr;
        setpriority(which, who, prio)
        int which, who, prio;
```

Description
The scheduling priority of the process, process group, or user, as indicated by **which** and **who** is obtained with the **getpriority** system call and set with the **setpriority** system call. The **which** argument is one of **PRIO_PROCESS**, **PRIO_PGRP**, or **PRIO_USER**, and **who** is interpreted relative to **which** (a process identifier for **PRIO_PROCESS**, a process group identifier for **PRO_PGRP**, and a user ID for **PRIO_USER**). A 0 value for **who** denotes the current process, process group, or user. The **prio** value is in the range 0 to 39. The default priority is 20; lower priorities cause more favorable scheduling.

A process's priority value is also referred to as its **nice** value.

The **getpriority** system call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The **setpriority** system call sets the priorities of all of the specified processes to the specified value. Only the superuser may lower priorities.

The **nice** system call adds the value of the **incr** parameter to the current process's priority value. If **incr** causes the priority value to fall outside the range 0 to 39, **nice** sets the priority value to the corresponding limit.

Return Value
Upon successful completion, **getpriority** and **nice** return a priority value, and **setpriority** returns 0. Otherwise, -1 is returned and *errno* is set to indicate the error.

Error Conditions
The **getpriority**, **setpriority** and **nice** system calls fail if one or more of the following are true:

ESRCH No process was located using the **which** and **who** values specified.

EINVAL **which** was not one of **PRIO_PROCESS, PRIO_PGRP,** or **PRIO_USER**.

EPERM The calling process does not have an effective user ID of the superuser.

EPERM The **incr** parameter is negative or greater than 40, and the

effective user ID of the calling process is not a superuser.

EACCES A non-superuser attempted to lower a process priority.

Subtopics
1.2.111.1 Compatibility Note

1.2.111.1 Compatibility Note

The routines **getpriority**, **setpriority**, and **nice** are also provided as compatibility routines in **libbsd.a**. The **libbsd.a** version of these routines differ from the **libc.a** version only in that they use priority values in the range of -20 to 20, instead of 0 to 39.

Note also that certain AIX commands with 4.3BSD origins (**csh**, **renice**, and the 4.3BSD mode of **ps**) display and expect priority values in the BSD range, while others (**nice** and the System V UNIX mode of **ps**) use priority values in the range of 0 to 39.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71 and " fork, vfork" in topic 1.2.83.

The **nice** command in *AIX Operating System Commands Reference*.

1.2.112 getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoer

Purpose
Gets protocol entry.

Library
Internet Library (**libc.a**)

Syntax
#include <netdb.h>

```
struct protoent *getprotoentvoid setprotoent (stayopen)
                                int stayopen;
struct protoent *getprotobyname (name)
char *name;                     void endprotoent ( )
struct protoent *getprotobynumber (proto)
int proto;
```

Description
The **getprotoent**, **getprotobyname**, and **getprotobynumber** subroutines each return a pointer to an object. This object is a **protoent** structure, which contains the field of a line in the **/etc/protocols** file (the network protocol data base). The **protoent** structure is defined in the **netdb.h** header file, and it contains the following members:

```
char    *p_name;        /* official name of protocol */
char    **p_aliases;    /* alias list                */
long    p_proto;        /* protocol number           */
```

The members of the structure are defined below:

p_name Official name of the protocol.

p_aliases An array, terminated by a 0, of alternate names for the protocol.

p_proto The protocol number.

The **getprotoent** subroutine reads the next line of the file. If the file is not open, **getprotoent** opens it.

The **setprotoent** subroutine opens and rewinds the file. If the **stayopen** parameter is 0, the protocol data base is closed after each call to **getprotobyname** or **getprotobynumber**. Otherwise, the file is not closed after each call.

The **endprotoent** subroutine closes the file.

The **getprotobyname** and **getprotobynumber** subroutines search the file sequentially from its beginning until finding a matching protocol name or protocol number, or until encountering the end of the file.

Return Value

A pointer to a **protoent** structure is returned on success.

Note: The return value points to static data that is overwritten by subsequent calls.

A NULL pointer (0) is returned if an error occurs or the end of the file is reached.

File

/etc/protocols Protocol name data base.

Related Information

The discussion of **/etc/protocols** in *AIX TCP/IP User's Guide*.

1.2.113 getpw

Purpose
Gets a password file entry, given the user ID.

Library
Standard C Library (**libc.a**)

Syntax

```
int getpw (uid, buf)
int uid;
char *buf;
```

Description
The **getpw** subroutine is included only for compatibility with prior systems and should not be used unless your program is going to be used with a prior system. See "getpwent, getpwuid, getpwnam, setpwent, endpwent" in topic 1.2.114 and "putpwent" in topic 1.2.215 for the correct subroutines to use.

The **getpw** searches the password file for a user ID number that matches the **uid** parameter. When a match is found, **getpw** copies the line of the password file in which the match was found into an array pointed to by the **buf** parameter. The subroutine then returns a value of 0. If a match cannot be found, the subroutine returns a nonzero value.

File
/etc/passwd

Related Information
In this book: "passwd" in topic 2.3.44.

1.2.114 getpwent, getpwuid, getpwnam, setpwent, endpwent

Purpose
Gets a password file entry.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <pwd.h>



struct passwd *getpwent ( ) void setpwent ( )

struct passwd *getpwuid (uidvoid endpwent ( )
int uid;

struct passwd *getpwnam (name)
char *name;
```

Description
The **getpwent**, **getpwuid**, and **getpwnam** subroutines return a pointer to a structure containing the broken-out fields of a line in the **/etc/passwd** file. The **passwd** structure is defined in the **pwd.h** header file, and it contains the following members:

```
char  *pw_name;
char  *pw_passwd;
uid_t pw_uid;
gid_t pw_gid;
char  *pw_age;
int   pw_quota;
char  *pw_comment;
char  *pw_etc;
char  *pw_dir;
char  *pw_shell;
```

The fields have meanings described in "passwd" in topic 2.3.44.

The **getpwent** subroutine, when first called, returns a pointer to the first **passwd** structure in the file. On the next call, it returns a pointer to the next **passwd** structure in the file. Successive calls can be used to search the entire file.

The **getpwuid** subroutine searches from the beginning of the file until it finds a numerical user ID matching the **uid** parameter. The subroutine then returns a pointer to the structure in which it was found.

The **getpwnam** subroutine searches from the beginning of the file until it finds a login name matching the **name** parameter. The search is made using ***flattened*** names; the characters of the name searched for are the ASCII equivalent character (see "Introduction to International Character Support" in *Managing the AIX Operating System*.) The subroutine then returns a pointer to the structure in which it was found.

If an end-of-file condition or an error is encountered on reading, these functions return a NULL pointer.

The **setpwent** subroutine rewinds the password file to allow repeated searches.

The **endpwent** subroutine closes the group file when processing is complete.

Warning: All information is contained in a static area, so it must be copied if it is to be saved.

File

/etc/passwd

Error Conditions
The **getpwuid** and **getpwnam** subroutines fail if one or more of the following are true:

EIO An I/O error has occurred.

EINTR A signal was caught during the function.

EMFILE Too many file descriptors are currently open for the process.

ENFILE The system file table is full.

Related Information
In this book: "getgrent, getgrgid, getgrnam, setgrent, endgrent" in topic 1.2.96, "getlogin" in topic 1.2.103, and "putpwent" in topic 1.2.215.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

1.2.115 getrlimit, setrlimit, vlimit

Purpose
Controls maximum system resource consumption.

Syntax

```
#include <sys/time.h>
#include <sys/resource.h>

        getrlimit (resource, rlp)   setrlimit (resource, rlp)
        int resource;               int resource;
        struct rlimit *rlp;         struct rlimit *rlp;
```

Description
Limits on the consumption of system resources by the current process and each process it creates may be obtained with the **getrlimit** call and set with the **setrlimit** call.

The **resource** parameter is one of the following:

RLIMIT_CPU The maximum amount of CPU time (in seconds) to be used by each process.

RLIMIT_FSIZE The largest size, in bytes, of any single file that may be created.

RLIMIT_DATA The maximum size, in bytes, of the data segment for a process; this defines how far a program may extend its break with the **sbrk** system call.

RLIMIT_STACK The maximum size, in bytes, of the stack segment for a process; this defines how far a program's stack segment may be extended. Stack extension is performed automatically by the system.

RLIMIT_CORE The largest size, in bytes, of a **core** file that may be created.

RLIMIT_RSS The maximum size, in bytes, to which a process's resident set size may grow. This imposes a limit on the amount of physical memory to be given to a process; if memory is tight, the system will prefer to take memory from processes that are exceeding their declared resident set size.

A resource limit is specified as a soft limit and a hard limit. When a soft limit is exceeded, a process may receive a signal (for example, if the CPU time is exceeded), but it will be allowed to continue execution until it reaches the hard limit (or modifies its resource limit). The **rlimit** structure is used to specify the hard and soft limits on a resource.

```
  struct rlimit {
 long   rlim_cur; /*current(soft)limit*/
 long   rlim_max; /*hard limit*/
  };
```

Only the superuser may raise the maximum limits. Other users may only alter **rlim_cur** within the range from 0 to **rlim_max** or (irreversibly) lower

rlim_max.

An infinite value for a limit is defined as **RLIM_INFINITY** (0x7fffffff).

Because this information is stored in the per-process information, this system call must be executed directly by the shell if it is to affect all future processes created by the shell; **limit** is thus a built-in command to **csh**.

The system refuses to extend the data or stack space when the limits would be exceeded in the normal way; a **break** call fails if the data space limit is reached. When the stack limit is reached, the process receives a segmentation fault (**SIGSEGV**). If this signal is not caught by a handler using the signal stack, this signal will kill the process.

When the soft CPU time limit is exceeded, a signal **SIGXCPU** is sent to the offending process.

Compatibility Note

To maintain upward compatibility with older BSD programs, the **vlimit** interface is also supported. It is used by compiling with the Berkeley Compatibility Library (**libbsd.a**). Its syntax is as follows:

#include <**sys/vlimit.h**>

vlimit (*resource*, *value***)**
int *resource*, *value*;

The flags for the *resource* parameter are defined in **sys/vlimit.h**, and are mapped to corresponding flags for **setrlimit**. The *value* parameter is an integer which is used as a hard limit parameter to **setrlimit**.

Return Value
A return value of 0 indicates that the call succeeded, changing or returning the resource limit. A return value of -1 indicates that an error occurred, and an error code is stored in the global location **errno**.

Error Conditions
The possible errors are:

EFAULT The address specified for **rlp** is invalid.

EPERM The limit specified to **setrlimit** would have raised the maximum limit value, and the caller is not the superuser.

Related Information
In this book: "setquota" in topic 1.2.253, "sigaction, sigvec, signal" in topic 1.2.263, and "sigstack" in topic 1.2.268.

The **csh** command in *AIX Operating System Commands Reference*.

1.2.116 getrusage, vtimes

Purpose
Gets information about resource utilization.

Syntax

```
#include <sys/time.h>
#include <sys/resource.h>

#define 0U/*calling process*/
#de-1ne /*terminatedRchild process*/

getrusage (who,rusage)
int who;
struct rusage *rusage;
```

Description
The **getrusage** system call returns information describing the resources utilized by the current process, or all its waited-for terminated child processes. The **who** parameter is one of **RUSAGE_SELF** or **RUSAGE_CHILDREN**. The buffer to which **rusage** points will be filled in with the following structure:

```
  struct rusage {
struct timeval ru_utime;    /*user time used*/
struct timeval ru_stime;    /*system time used*/
long  ru_maxrss;
long  ru_ixrss;             /*integral shared text memory size*/
long  ru_idrss;             /*integral unshared data size*/
long  ru_isrss;             /*integral unshared stack size*/
long  ru_minflt;            /*page reclaims*/
long  ru_majflt;            /*page faults*/
long  ru_nswap;             /*swaps*/
long  ru_inblock;           /*block input operations*/
long  ru_oublock;           /*block output operations*/
long  ru_msgsnd;            /*messages sent*/
long  ru_msgrcv;            /*messages received*/
long  ru_nsignals;          /*signals received*/
long  ru_nvcsw;             /*voluntary context switches*/
long  ru_nivcsw;            /*involuntary context switches*/
  };
```

The fields are interpreted as follows:

ru_utime The total amount of time spent executing in user mode.

ru_stime The total amount of time spent in the system executing on behalf of the process(es).

ru_maxrss The maximum resident set size utilized (in kilobytes).

ru_ixrss An integral value indicating the amount of memory used by the text segment that was also shared among other processes. This value is expressed in units of kilobytes * seconds-of-execution and is calculated by summing the number of shared memory pages in use each time the internal system clock ticks and then averaging over one second intervals.

ru_idrss An integral value of the amount of unshared memory residing in

the data segment of a process (expressed in units of kilobytes * seconds-of-execution).

ru_isrss An integral value of the amount of unshared memory residing in the stack segment of a process (expressed in units of kilobytes * seconds-of-execution).

ru_minflt The number of page faults serviced without any I/O activity; here I/O activity is avoided by reclaiming a page frame from the list of pages awaiting reallocation.

ru_majflt The number of page faults serviced that required I/O activity.

ru_nswap The number of times a process was swapped out of main memory.

ru_inblock The number of times the file system had to perform input.

ru_oublock The number of times the file system had to perform output.

ru_msgsnd The number of IPC messages sent.

ru_msgrcv The number of IPC messages received.

ru_nsignals The number of signals delivered.

ru_nvcsw The number of times a context switch resulted due to a process voluntarily giving up the processor before its time slice was completed (usually to await availability of a resource).

ru_nivcsw The number of times a context switch resulted due to a higher priority process becoming runnable or because the current process exceeded its time slice.

Note: The numbers **ru_inblock** and **ru_oublock** account only for real I/O; data supplied by the caching mechanism is charged only to the first process to read or write the data.

Compatibility Note

To provide compatibility with older programs, the interface to the BSD **vtimes** function is supported. It is used by compiling with the Berkeley Compatibility Library (**libbsd.a**). Its syntax is as follows:

```
#include <sys/vtimes.h>

vtimes (par_vm, ch_vm)
struct vtimes * par_vm, *ch_vm;
```

The **vtimes** subroutine returns accounting information for the current process and for the terminated child processes of the current process. Either *par_vm* or *ch_vm* or both may be 0, in which case only the information for the pointers which are nonzero is returned.

After the call, each buffer contains information as defined by the contents of the include file **sys/vtimes.h**.

Error Conditions
The possible errors for **getrusage** are:

EINVAL The **who** parameter is not a valid value.

EFAULT The address specified by the **rusage** parameter is not in a valid part of the process address space.

Related Information
In this book: "gettimeofday, settimeofday, ftime" in topic 1.2.123 and "wait, waitpid" in topic 1.2.325.

1.2.117 gets, fgets, getws, fgetws

Purpose
Reads characters or wide characters from a stream.

Library
Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>

char *gets (s)
char *s;

wchar_t *getws (s)
wchar_t *s;

wchar_t *fgetws (s, n, stream)
wchar_t *s;
int n;
FILE *stream;

char *fgets (s, n, stream)
char *s;
int n;
FILE *stream;
```

Description
The **gets** subroutine reads characters from the standard input stream, **stdin**, into the array pointed to by the **s** parameter. Data is read until a new-line character is read or an end-of-file condition is encountered. If reading is stopped due to a new-line character, the new-line character is discarded and the string is terminated with a null character.

The **fgets** subroutine reads characters from the data pointed to by the **stream** parameter into the array pointed to by the **s** parameter. Data is read until **n** - 1 characters have been read, until a new-line character is read and transferred to **s**, or until an end-of-file condition is encountered. The string is then terminated with a null character.

The **getws** subroutine reads wide characters from the input stream pointed to by **stdin** into the array pointed to by **s** until end-of-file is encountered or a new-line character is read. Any new-line character is discarded, and a NULL character is written immediately after the last character read into the array.

The **fgetws** subroutine reads wide characters (at most one less than the number of the characters specified by **n**) from the stream pointed to by **stream** into the array pointed to by **s**. No additional characters are read after a new-line character (which is retained) or after end-of-file. A NULL character is written immediately after the last character read into the array.

Return Value
If end-of-file is encountered and no characters have been read, no characters are transferred to **s** and a NULL pointer is returned. If a read error occurs, a NULL pointer is returned. Otherwise, **s** is returned.

Error Conditions
The **gets** and **fgets** subroutines fail if one or more of the following are true:

EAGAIN The **O_NONBLOCK** flag is set for the file descriptor underlying **stream** and the process would be delayed in the **fgetc** operation.

EBADF The file descriptor underlying **stream** is not a valid file descriptor open for reading.

EINTR The read operation was terminated due to the receipt of a signal and no data was transferred.

EIO The process is a member of a background process attempting to read from its controlling terminal, the process is either ignoring or blocking the **SIGTTIN** signal or the process group is orphaned.

ENOMEM Insufficient storage space is available.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device.

Related Information
In this book: "feof, ferror, clearerr, fileno" in topic 1.2.79, "fopen, freopen, fdopen" in topic 1.2.82, "fread, fwrite" in topic 1.2.84, "getc, fgetc, getchar, getw, getwc, fgetwc, getwchar" in topic 1.2.91, "puts, fputs, putws, fputws" in topic 1.2.216, "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf, wsscanf" in topic 1.2.241, and " stdio" in topic 1.2.283.

1.2.118 getservent, getservbyname, getservbyport, setservent, endservent

Purpose
Gets service entry.

Library
Internet Library (**libc.a**)

Syntax
#include <netdb.h>

```
struct servent *getservent ( )  void setservent (stayopen)
                                int stayopen;
struct servent *getservbyname (name, proto)
char *name, *proto;             void endservent ( )
struct servent *getservbyport (port, proto)
int port;
char *proto;
```

Description
The **getservent**, **getservbyname**, and **getservbyport** subroutines each return a pointer to an object. This object is a **servent** structure, which contains the field of a line in the **/etc/services** file (the network services data base). The **servent** structure is defined in the **netdb.h** header file, and it contains the following members:

```
char    *s_name;      /* official name of service   */
char    **s_aliases;  /* alias list                 */
long    s_port;       /* port where service resides */
char    *s_proto;     /* protocol to use            */
```

The members of the structure are defined below:

s_name Official name of the service.

s_aliases An array, terminated by a 0, of alternate names for the service.

s_port The port number at which the service resides. Port numbers are returned in network byte order.

s_proto The name of the protocol to use when contacting the service.

The **getservent** subroutine reads the next line of the file. If the file is not open, **getservent** opens it.

The **setservent** subroutine opens and rewinds the file. If the **stayopen** parameter is 0, the service data base is closed after each call to **getservbyname** or **getservbyport**. Otherwise, the file is not closed after each call.

The **endservent** subroutine closes the file.

The **getservbyname** and **getservbyport** subroutines search the file sequentially from its beginning until finding a matching protocol name or port number, or until encountering the end of the file. When a protocol name is also supplied, searches also match the protocol.

Return Value

A pointer to a **servent** structure is returned on success.

Note: The return value points to static data that is overwritten by subsequent calls.

A NULL pointer (0) is returned if an error occurs or the end of the file is reached.

File

/etc/services Service name data base.

Related Information

In this book: "getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent" in topic 1.2.112.

The discussion of **/etc/services** in *AIX TCP/IP User's Guide*.

1.2.119 getsites

Purpose
Determines sites which are in the current TCF cluster.

Syntax

```
#include <sys/types.h>

int getsites (sitep, maxsites)
sitestat_t *sitep;
int maxsites;
```

Description
The **getsites** system call returns information about which AIX/370 and AIX PS/2 sites are in the current TCF cluster. These sites are also referred to collectively as the current network partition. The **sitep** argument is a pointer to a buffer of length **maxsites*sizeof** (**sitestat_t**), with each element representing a site (that is, **sitep**[i] is the status information for site **i**). Site **i** is in the current TCF cluster if (**sitep[i]** & **GS_UP**) **!= 0**. If **maxsites** is not large enough to return all of the site information, -1 is returned. Otherwise, the return value is the maximum site number plus one. A good value to be passed as the **maxsites** argument is the constant MAXSITE which is defined in the file **<sys/param.h>**.

Return Value
Upon successful completion of **getsites**, the maximum site number plus one is returned. Otherwise, -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **getsites** system call fails if any of the following are true:

EINVAL **maxsites** is zero or negative.

EFAULT **sitep** does not point to a writable region inside the user's address space.

EFAULT **maxsites** is not big enough to return all the site information.

Related Information
In this book: "netctrl" in topic 1.2.185, and "site" in topic 1.2.272.

The **clusterstart** and **clusterstop** commands in the **AIX Operating System Command Reference**.

1.2.120 getsockname

Purpose
Gets the socket name.

Library
Internet Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockname (s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

Description

The **getsockname** system call stores the current name for the socket specified by the **s** parameter in the structure pointed to by the **name** parameter. Initialize the value pointed to by the **namelen** parameter to indicate the amount of space pointed to by **name**. On return, the **namelen** parameter points to the actual size (in bytes) of the name returned (in bytes).

Return Value

Upon successful completion, a value of 0 is returned. If the **getsockname** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The system call fails if one or more of the following are true:

EBADF The **s** parameter is not valid.

ENOTSOCK The **s** parameter refers to a file, not a socket.

ENOBUFS Insufficient resources were available in the system to complete the call.

EFAULT The **addr** parameter is not in a writable part of the user address space.

Related Information
In this book: "bind" in topic 1.2.20 and "socket" in topic 1.2.275.

1.2.121 getsockopt, setsockopt

Purpose
Gets and sets options on sockets.

Library
Internet Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt (s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int *optlen;

int setsockopt (s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int optlen;
```

Description

The **getsockopt** and **setsockopt** system calls manipulate options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost socket level.

When manipulating socket options, you must specify the level at which the option resides and the name of the option. To manipulate options at the socket level, specify **level** as **SOL_SOCKET**. To manipulate options at any other level, supply the appropriate protocol number for the protocol controlling the option. For example, to indicate that an option will be interpreted by the TCP protocol, set **level** to the protocol number of TCP. For more information, see "getprotoent" on page 1.2.112.

Use the parameters **optval** and **optlen** to access option values for **setsockopt**. For **getsockopt**, these parameters identify a buffer in which the value for the requested option or options are returned. For **getsockopt**, the **optlen** parameter initially contains the size of the buffer pointed to by the **optval** parameter. On return, the **optlen** parameter is modified to indicate the actual size of the value returned. If no option value is supplied or returned, the **optval** parameter can be 0.

Most socket-level options take an **int** parameter for **optval**. For **setsockopt**, the parameter should be nonzero to enable a boolean option, or 0 if the option is to be disabled. **SO_LINGER** uses a **struct linger** parameter, defined in **sys/socket.h**, which specifies the desired state of the option and the linger interval.

The **optname** parameter and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The **sys/socket.h** header file contains definitions for socket level options. These options are:

SO_DEBUG Turns on recording of debugging information.
SO_REUSEADDR Allows local address reuse.
SO_KEEPALIVE Keeps connections active.
SO_DONTROUTE Does not apply routing on outgoing messages.

SO_LINGER	Lingers on a **close** system call if data is present.
SO_OOBINLINE	Leaves received ***out-of-band data*** (data marked urgent) in line.
SO_SNDBUF	Sends buffer size.
SO_RCVBUF	Receives buffer size.
SO_ERROR	Gets error status.
SO_TYPE	Gets socket type.
SO_BROADCAST	Request permission to transmit broadcast messages.

SO_DEBUG enables debugging in the underlying protocol modules. **SO_REUSEADDR** indicates that the rules used in validating addresses supplied by a **bind** system call should allow reuse of local addresses. **SO_KEEPALIVE** enables the periodic transmission of messages on a connected socket. If the connected socket fails to respond to these messages, the connection is broken and processes using that socket are notified with a **SIGPIPE** signal. **SO_DONTROUTE** indicates that outgoing messages should bypass the standard routing facilities and are directed to the appropriate network interface according to the network portion of the destination address. **SO_LINGER** controls the action taken when unsent messages are queued on a socket and a **close** system call is performed. If **SO_LINGER** is set, the system blocks the process during the **close** system call until it can transmit the data or until the time expires. Specify the amount of time for the linger interval by using the **setsockopt** system call when requesting **SO_LINGER**. If **SO_LINGER** is not specified and a **close** system call is issued, the system handles the call in a way that allows the process to continue as quickly as possible.

The option **SO_BROADCAST** requests permission to send broadcast datagrams on the socket. With protocols that support out-of-band data, the **SO_OOBINLINE** option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with **recv** or **read** system calls without the **MSG_OOB** flag. **SO_SNDBUF** and **SO_RCVBUF** are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values. Finally, **SO_TYPE** and **SO_ERROR** are options used only with **setsockopt**. **SO_TYPE** returns the type of the socket, such as **SOCK_STREAM**; it is useful for servers that inherit sockets on startup. **SO_ERROR** returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

Options at other protocol levels vary in format and name.

Return Value

Upon successful completion, a value of 0 is returned. If the **getsockopt** or **setsockopt** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The system calls fail if one or more of the following are true:

EBADF	The **s** parameter is not valid.
ENOTSOCK	The **s** parameter refers to a file, not a socket.
ENOPROTOOPT	The option is unknown.

EFAULT The **optval** parameter is not in a writable part of the user address space.

Related Information

In this book: "getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent" in topic 1.2.112, and "socket" in topic 1.2.275.

1.2.122 getspath, setspath

Purpose
Manages the site path list.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>

getspath(site_path, length)
sitepath_t *site_path;
int length;

setspath(site_path, length)
sitepath_t *site_path;
int length;
```

Description
The site path is used when resolving file names which refer to hidden directories. It is also used to select execution sites in the **exec**, **rexec**, **run**, and **migrate** system calls.

The site path consists of an array of site path elements. These elements may be of three possible types:

NULLSITE Refers to the local site or to that site's machine type or a compatible type.

site_number
The number of a specific cluster site. This type of element refers to the specified site or to that site's machine type or a compatible type.

(cpu_type|SPATH_CPU)
A machine type number that is specified in the file **<a.out.h>**. It refers to the specified machine type or to an arbitrarily selected site of that CPU type.

NULLSITE and **SPATH_CPU** are constants defined in the file **<sys/types.h>**. Known **cpu_type** numbers include:

code	**common name**	**hidden directory component**	**compatible type**
CPU_386	iAPX 80386	i386	none
CPU_S370	System/370	i370	none
CPU_XA370	XA/370	xa370	i370

When selecting a component of a hidden directory (see "chhidden" in topic 1.2.42), each element of the site path is tried in turn. If a file is found in the hidden directory with the name of the machine type or the name of a compatible type referred to by the site path element, that file is used. A compatible type is chosen only if the exact type is not found.

The site path is also used to choose a site for execution in the case of an **exec** system call or in the case of a **migrate**, **rexec**, or **run** system call which is passed a **site_number** argument of 0. In these cases, the site is

chosen as follows. First, the system determines the machine type on which the new process file (for **exec**, **rexec**, or **run**) or the current process (for **migrate**) must run. Then, the site path is searched until an element is found which corresponds to that machine type. If the element is **NULLSITE**, the process runs locally. If the element is a site number, the process executes on that site. If the element has the **SPATH_CPU** bit turned on, the process runs on a randomly chosen site of the specified type (except that the local site is chosen if it is the right machine type).

In a **setspath** call, the **site_path** argument points to an array of **length** elements of type **sitepath_t**. In a **getspath** call, the **site_path** argument points to an array of **sitepath_t** elements into which the system will return at most **length** values.

Return Value

Upon successful completion, **setspath** returns a value of 0 to the calling process and **getspath** returns the length of the **site_path** array to the calling process. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions

These calls fail if any of the following are true:

EINVAL **length** is greater than the maximum site path length allowed by the system (**setspath** only).

EINVAL **length** is less than the length of the current process's site path (**getspath** only).

EFAULT **site_path** points to an invalid address.

EBADST The site path specified by **site_path** contains an invalid site or **cpu_type** number (**setspath** only).

Related Information

In this book: "chhidden" in topic 1.2.42, "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, "migrate" in topic 1.2.167, " rexec: rexecl, rexecv, rexecle, rexecve, rexeclp, rexecvp" in topic 1.2.236, and " run: runl, runv, runle, runve, runlp, runvp" in topic 1.2.239.

1.2.123 gettimeofday, settimeofday, ftime

Purpose
Obtains current time.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/time.h>

int gettimeofday (tp, tzp)
struct timeval *tp;
struct timezone *tzp;

int settimeofday (tp, tzp)
struct timeval *tp;
struct timezone *tzp;

int ftime  (tp)
struct timeb *tp;
```

Description
The **gettimeofday** system call gets the current Greenwich time and the current time zone. The **settimeofday** system call sets the time and the time zone. The time is expressed in seconds from 00:00:00 GMT January 1, 1970. If **tzp** equals 0, the time zone is neither returned nor set.

The **tp** parameter points to a **timeval** structure, defined in the file **sys/time.h**. This structure contains the following members:

```
long  tv_sec;          /* Seconds since Jan. 1, 1970     */
long  tv_usec;         /* Microseconds to add to seconds */
```

The **tzp** parameter points to a **timezone** structure, defined in the **sys/time.h** file. This structure contains the following members:

```
int   tz_minuteswest;  /* Time west of Greenwich in minutes */
int   tz_dsttime;      /* Type of DST correction to apply */
```

The **timezone** structure indicates the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that daylight saving time applies locally during the appropriate part of the year.

The *tp* parameter returns a pointer to a structure which contains the time since the epoch in seconds, up to 1000 milliseconds of more precise interval, the local time zone (measured in minutes westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

Only the superuser may set the time of day or time zone.

Compatibility Note

The **ftime** subroutine is included for compatibility with older BSD programs. Its function has been made obsolete by **gettimeofday**. It is used by compiling with Berkeley Compatibility Library (**libbsd.a**).

Return Value
If the call succeeds, a value of 0 is returned. If an error occurs, a value of -1 is returned, and an error code is placed in the global variable **errno**.

Error Conditions
The subroutines fail if one or more of the following are true:

EFAULT The **tz** or **tzp** parameter is not in a writable part of the user address space.

EPERM A user other than the superuser attempted to set the time.

Related Information
In this book: "ctime, localtime, gmtime, asctime, tzset" in topic 1.2.54 and "time" in topic 1.2.303.

The **date** command in *AIX Operating System Commands Reference*.

The discussion of **timed** in *AIX TCP/IP User's Guide*.

1.2.124 getuid, geteuid, getgid, getegid

Purpose
Gets the real user, effective user, real group, and effective group IDs.

Library
Standard C Library (**libc.a**)

Syntax

```
uid_t getuid ( )          uid_t getgid ( )

uid_t geteuid ( )         uid_t getegid ( )
```

Description
The **getuid** system call returns the real user ID of the calling process.

The **geteuid** system call returns the effective user ID of the calling process.

The **getgid** system call returns the real group ID of the calling process.

The **getegid** system call returns the effective group ID of the calling process.

Related Information
In this book: "setuid, setgid" in topic 1.2.255.

1.2.125 getuinfo

Purpose
Finds the value associated with a user information name.

Library
Standard C Library (**libc.a**)

Syntax

```
char *getuinfo (name)
char *name;
```

Description
The **getuinfo** subroutine searches a user information buffer for a string of the form **name=value** and returns a pointer to the **value** substring if **name** is found. NULL is returned if **name** is not found.

The user information buffer searched is pointed to by the global variable:

```
extern char *INuibp;
```

This variable is initialized to NULL.

If the variable **INuibp** is NULL when the **getuinfo** subroutine is called, the **usrinfo** system call is executed to read user information from the kernel into a local buffer. The address of the buffer is then put into the external variable **INuibp**. The **usrinfo** system call is automatically called the first time the **getuinfo** subroutine is called if the **INuibp** variable has not been set.

Related Information
In this book: "usrinfo" in topic 1.2.319.

1.2.126 getut: getutent, getutid, getutline, pututline, setutent, endutent, ut

Purpose
Accesses **utmp** file entries.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <utmp.h>



struct utmp *getutent ( )   void pututline (utmp)
                            struct utmp *utmp;
struct utmp *getutid (id)
struct utmp *id;            void setutent ( )

struct utmp *getutline (linevoid endutent ( )
struct utmp *line;
                            void utmpname (file)
                            char *file;
```

Description
The **getutent**, **getutid**, and **getutline** subroutines each return a pointer to a structure of the following type:

```
#define ut_name  ut_user

struct utmp
  {
     char ut_user[8];          /* User login name */
     char ut_id[6];            /* id from /etc/inittab */
     char ut_line[12];         /* device name (console, ttyx) */
     pid_t ut_pid;             /* process id */
     short  ut_type;           /* type of entry */
     struct exit_status
       {
        short e_termination;   /* Process termination status */
        short e_exit;          /* Process exit status */
       }
     ut_exit;                  /* The exit status of a process */
                               /* marked as DEAD_PROCESS. */
     time_t  ut_time;          /* time entry was made */
     char   ut_host[16];       /* host name if remote login */
     long   ut_lsite;          /* reserved */
     char ut_datarep[4];       /* reserved */
};
```

The **getutent** subroutine reads the next entry from a **utmp**-like file. If the file is not already open, this subroutine opens it. If the end of the file is reached, **getutent** fails.

If you specify type **RUN_LVL**, **BOOT_TIME**, **OLD_TIME**, or **NEW_TIME** in the **id** parameter, the **getutid** subroutine searches forward from the current point in the **utmp** file until an entry with a **ut_type** matching **id->ut_type** is found.

If you specify one of the types **INIT_PROCESS**, **LOGIN_PROCESS**, **USER_PROCESS**

or **DEAD_PROCESS** in the **id** parameter, then the **getutid** subroutine returns a pointer to the first entry whose type is one of these four and whose **ut_id** field matches **id->ut_id**. If the end of the file is reached without a match, the **getutid** subroutine fails.

The **getutline** subroutine searches forward from the current point in the **utmp** file until it finds an entry of the type **LOGIN_PROCESS** or **USER_PROCESS** that also has a **ut_line** string matching the **line->ut_line** parameter string. If the end the of file is reached without a match, the **getutline** subroutine fails.

The **pututline** subroutine writes the supplied **utmp** structure into the **utmp** file. If you have not searched for the proper place in the file using one of the **getut** routines, then the **pututline** subroutine calls **getutid** to search forward for the proper place. It is expected that normally the user of **pututline** searched for the proper entry using one of the **getut** subroutines. If so, **pututline** does not search. If the **pututline** subroutine does not find a matching slot for the entry, it adds a new entry to the end of the file.

The **setutent** subroutine resets the input stream to the beginning of the file. You should do this before each search for a new entry if you want to examine the entire file.

The **endutent** subroutine closes the currently open file.

The **utmpname** subroutine changes the name of the file to be examined from **/etc/utmp** to any other file. The name specified is usually **/usr/adm/wtmp**. If the specified file does not exist, no indication is given. You are not aware of this fact until your first attempt to reference the file. The **utmpname** subroutine does not open the file. It closes the old file, if it is currently open, and saves the new file name.

The most current entry is saved in a static structure. If you desire to make multiple accesses, you must copy or use the structure between each access. The **getutid** and **getutline** subroutines examine the static structure first. If the contents of the static structure match what they are searching for, they do not read the **utmp** file. Therefore, you must fill the static structure with zeros after each use if you want to use these subroutines to search for multiple occurrences.

If **pututline** finds that it isn't already at the correct place in the file, then the implicit read it performs does not overwrite the contents of the static structure returned by the **getutent**, **getuid**, or **getutline** routine. This allows you to get an entry with one of these subroutines, modify the structure, and pass the pointer back to **pututline** for writing.

These subroutines use buffered standard I/O for input, but **pututline** uses an unbuffered nonstandard write to avoid race conditions between processes trying to modify the **utmp** and **wtmp** files.

Return Value
These subroutines fail and return a NULL pointer if a read or write fails due to end-of-file or a permission conflict.

Files

/etc/utmp
/usr/adm/wtmp

Related Information

In this book: "ttyslot" in topic 1.2.312 and "utmp, wtmp, .ilog" in topic 2.3.60.

1.2.127 getwd

Purpose
Gets current directory path name.

Library
Standard C Library (**libc.a**)

Syntax

```
char *getwd (pathname)
char *pathname;
```

Description
The **getwd** subroutine determines the absolute path name of the current directory, then copies that path name into the area pointed to by the **pathname** parameter.

The maximum path name length, in characters, is set by the **MAXPATHLEN** define directive in the file **/usr/include/sys/param.h**.

Return Value
If the call to **getwd** is successful, a pointer to the absolute path name of the current directory is returned. If an error occurs, **getwd** returns the value 0 and places a message in **pathname**.

Related Information
In this book: "getcwd" in topic 1.2.92.

1.2.128 getxperm, setxperm

Purpose
Manages user's execution permission site mask.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>

int getxperm(sitep,maxsites)
sitexperm_t *sitep;
int maxsites;

int setxperm(sitep,maxsites)
sitexperm_t *sitep;
int maxsites;

#define GS_NOPERMISSION 0
#define GS_PERMISSION   1
```

Description
The **getxperm** system call retrieves the user site permission mask of the current process. The user site permission mask is the list of sites for which the user has execution/migration permission. The information is returned in the **sitep** buffer which is assumed to be of length **maxsites**. If the **sitep** buffer is of insufficient length to return all the site information, then an error occurs and -1 is returned. Successful completion is indicated by a return value equivalent to the maximum site number plus one.

The **setxperm** system call sets the user site permission mask of the current process. The arguments for **setxperm** are interpreted similarly to those for **getxperm**. Any sites not represented by **sitep** and **maxsites** remain unaltered. A successful call returns 0. A return value of -1 indicates an error. Only the superuser may issue **setxperm**.

The site permission mask indicates where the current process may move or create new processes. Permission is granted for those sites which are set to **GS_PERMISSION**. The site permission mask is inherited by the child processes when the process creates them.

Return Value
Upon successful completion of **setxperm**, a value of 0 is returned to the calling process. Upon successful completion of **getxperm**, a value equivalent to the maximum site number plus one is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **getxperm** and **setxperm** system calls fail if any of the following are true:

EINVAL **maxsites** is less than one or greater than 255 in a **setxperm** call. **maxsites** is less than 1 or less than the currently configured number of sites on the network or greater than 255 in a **getxperm** system call.

EFAULT **sitep** does not point to a valid area inside the user's address

space.

EFAULT The **sitep** buffer is of insufficient length to return all the site information in a **getxperm** call.

EPERM **setxperm** was called from a process whose effective user ID was not superuser.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, " fork, vfork" in topic 1.2.83, "migrate" in topic 1.2.167, " rexec: rexecl, rexecv, rexecle, rexecve, rexeclp, rexecvp" in topic 1.2.236, "rfork" in topic 1.2.237, and " run: runl, runv, runle, runve, runlp, runvp" in topic 1.2.239.

1.2.129 getxvers, setxvers

Purpose
Manages the execution version string for hidden directories.

Library
Standard C Library (**libc.a**)

Syntax

```
getxvers (xvers, length)
char *xvers;
int length;

setxvers (xvers)
char *xvers;
```

Description
The **getxvers** and **setxvers** system calls get and set the process's **xvers** string.

The **xvers** string is used to modify the hidden directory search path (see "getspath, setspath" in topic 1.2.122). It is a null-terminated string.

When searching to choose a component of a hidden directory, each element of the site path is used in turn. For each element of the site path, the system first searches for the **xvers** string, concatenated with the machine type name, then for just the machine type name.

To undo the **xvers** string, set it to a zero length string.

Return Value
These calls will fail if any of the following are true:

EINVAL The current process's **xvers** string (including the null character) is longer than the **length** characters (**getxvers** only).

EINVAL The string pointed to by **xvers** is longer than the system-imposed limit (**setxvers** only).

EFAULT **xvers** points to an invalid address.

Error Conditions
Upon successful completion, these calls will return a value of 0 to the calling process. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Related Information
In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, "getspath, setspath" in topic 1.2.122, and "open, openx, creat" in topic 1.2.199.

The **csh** and **sh** commands in *AIX Operating System Commands Reference*.

1.2.130 hsearch, hcreate, hdestroy

Purpose
Manages hash tables.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <search.h>


ENTRY *hsearch (item, actionint hcreate (nel)
ENTRY item;                 unsigned int nel;
ACTION action;
                            void hdestroy ( )
```

Description
The **hsearch** subroutine is a hash table search routine. It returns a pointer into a hash table that indicates the location of a given entry. The **item** parameter is a structure of the type ENTRY as defined in the **search.h** header file. It contains two pointers:

item.**key** Points to the comparison key.
item.**data** Points to any other data be associated with that key.

Pointers to types other than **char** should be cast to pointer-to-character. The **action** parameter is a value of the ACTION enumeration type that indicates what is to be done with an entry if it cannot be found in the table:

ENTER Enters the item into the table at the appropriate point. If the table is full, a NULL pointer is returned.

FIND Does not enter the item into the table, but returns a NULL pointer if the item cannot be found.

The **hsearch** subroutine uses open addressing with a multiplicative hash function.

The **hcreate** subroutine allocates sufficient space for the table. You must call **hcreate** before calling **hsearch**. The **nel** parameter is an estimate of the maximum number of entries that the table contains. Under some circumstances, **hcreate** may actually make the table larger than specified. Upon successful completion, **hcreate** returns 1. **hcreate** returns 0 if it cannot allocate sufficient space for the table.

The **hdestroy** subroutine deletes the hash table. This allows you to start a new hash table since only one table can be active at a time.

Related Information
In this book: "bsearch" in topic 1.2.23, "lsearch, lfind" in topic 1.2.160, "string" in topic 1.2.288, and "tsearch, tdelete, twalk" in topic 1.2.309.

1.2.131 htonl, htons, ntohl, ntohs

Purpose
Converts values between host and Internet network byte order.

Library
Internet Library **(libc.a)**

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>


unsigned long htonl (hostlonunsigned long ntohl (netlong)
unsigned long hostlong;     unsigned long netlong;

unsigned short htons (hostshunsigned short ntohs (netshort)
unsigned short hostshort;   unsigned short netshort;
```

Description
These subroutines convert 16- and 32-bit quantities between network byte order and host byte order. On machines already in network byte order (such as the IBM System/370) these routines are defined as null macros in the include file **netinet/in.h**.

These subroutines are often used in conjunction with Internet addresses and ports as returned by the **gethostent** and **getservent** subroutines.

Related Information
In this book: "gethostbyaddr, gethostbyname, sethostent, endhostent" in topic 1.2.98 and "getservent, getservbyname, getservbyport, setservent, endservent" in topic 1.2.118.

1.2.132 hypot, cabs

Purpose
Computes the euclidean distance function and complex absolute value.

Library
Math Library (**libm.a**)

Syntax

```
#include <math.h>

double hypot (x, y)
double x, y;

double cabs (z)
struct {double x,y} z
```

Description
The **hypot** and **cabs** subroutines take precautions against overflows while computing the value of:

If the correct value does overflow, then **hypot** returns HUGE and sets **errno** to ERANGE.

You can change the error-handling procedures by supplying a **matherr** subroutine. See "matherr" in topic 1.2.163 for more information.

Error Conditions
The **hypot** subroutine may fail if one or more of the following are true:

EDOM The value of **x** or **y** is NaN.

ERANGE The value to be returned would have caused overflow.

Related Information
In this book: "cbrt, exp, expm1, log, log10, log1p, pow, sqrt" in topic 1.2.28.

1.2.133 index, rindex

Purpose
Locates a character in a string.

Library
Standard C Library (**libc.a**)

Syntax

```
char *index (string, character)        char *rindex (string, character)
char *string, character;               char *string, character;
```

Description
The **index** subroutine returns a pointer to the first occurrence of **character** in **string**, while the **rindex** subroutine returns a pointer to the last occurrence of **character** in **string**. Both subroutines return a value of 0 if the character does not occur in the string.

These functions operate on null-terminated strings.

Because the BSD4.3 subroutines **index** and **rindex** have been implemented here as calls to the System V **strchr** and **strrchr** subroutines, be prepared for potential unexpected results if you are porting code using the **index** and **rindex** subroutines described here. See "string" in topic 1.2.288 for more information on the **strchr** and **strrchr** subroutines which can be used to perform the same operations.

Note also that the **index** subroutine described here is different from that available with Programmer's Workbench Library **(libPW.a)**. If you link from **libPW.a**, you cannot use the **index** subroutine; you must use the **strchr** subroutine instead.

Related Information
In this book: "string" in topic 1.2.288.

1.2.134 inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_ne

Purpose
Manipulation subroutines for Internet addresses.

Library
Internet Library (**libc.a**)

Syntax

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>

unsigned long inet_addr (cp)          struct in_addr inet_makeaddr (net,
char *cp;                             unsigned long net, lna;

unsigned long inet_network (          unsigned long inet_lnaof (in)
char *cp;                             struct in_addr in;

char *inet_ntoa (in)                  unsigned long inet_netof (in)
struct in_addr in;                    struct in_addr in;
```

Description
The **inet_addr** subroutine interprets a character string as a full Internet address in dot (.) notation and returns a number suitable for use as an Internet address. The **inet_network** subroutine interprets a character string as the network portion of an Internet address in dot notation, and returns the network number.

The **inet_ntoa** subroutine takes an Internet address and returns an ASCII string representing the address in dot notation. The **in** parameter contains the Internet address to be converted to ASCII.

The **inet_makeaddr** takes an Internet network number and a local network address and constructs an Internet address from it. The **net** parameter contains an Internet network number, while the **lna** parameter contains a local network address.

The **inet_netof** and **inet_lnaof** subroutines break apart Internet addresses, returning the network number and local network address part. The **in** parameter represents the Internet address to separate.

All Internet addresses are returned in network byte order. All network numbers and local addresses are returned as unsigned integer values in host order.
The values specified using the dot notation take one of the following forms:

a.b.c.d
a.b.c
a.b
a

The **inet_addr** subroutine interprets input strings in the following way.

When four parts are specified, each is interpreted as a byte of data and assigned to the four bytes of an Internet address.

When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right two bytes of the Internet address. This makes the three-part address format convenient for specifying Class B network addresses as **B0.B1.HOST**, where 128 <= B0 <= 191, 0 <= B1 <= 255, and 0 <= HOST <= 65535.

When a two-part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right three bytes of the Internet address. This makes the two-part address format convenient for specifying Class A network addresses as **A0.HOST**, where 0 <= A0 <= 127, and 0 <= HOST <= 16777215.

The interpretation of a one-part address is undefined; such addresses should not be passed to **inet_addr**.

The **inet_network** subroutine interprets input strings in the following way.

When a one part address is specified it is interpreted as a Class A address. When a two-part part address is specified it is interpreted as a Class B address. When a three-part part address is specified it is interpreted as a Class C address.

The interpretation of a four-part address is undefined; such addresses should not be passed to **inet_network**. To obtain the network number from a full Internet address, use the subroutine **inet_netof** in conjuction with **inet_addr**; for example, **net = inet_netof(inet_addr(cp))**.

All numbers supplied for each part of a dot notation may be decimal, octal, or hexadecimal, as specified in C language. A leading **0x** or **0X** implies hexadecimal, a leading **0** implies octal, and anything else is interpreted as decimal.

Return Value
The **inet_addr** and **inet_network** subroutines return numbers suitable for use as Internet addresses and Internet network numbers, respectively, on success.

If the **inet_addr** or **inet_network** subroutine fails, a value of (unsigned long) -1 is returned. To test for failure it is necessary to cast -1 to an unsigned long when comparing with the return value. Note that for **inet_addr** the value returned for addresses "255.255.255.255" and "255.255.255.254" is indistinguishable from (unsigned long) -1 on two's complement and one's complement machines respectively.

Related Information
In this book: "gethostbyaddr, gethostbyname, sethostent, endhostent" in topic 1.2.98, and "getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent" in topic 1.2.105.

The discussion of **/etc/hosts** and **/etc/networks** in *AIX TCP/IP User's Guide*.

1.2.135 initgroups

Purpose
Initializes group access list.

Library
Standard C Library (**libc.a**)

Syntax

```
int initgroups (user, basegid)
char  *user;
int basegid;
```

Description
The **initgroups** subroutine reads the **/etc/group** file and constructs the group access list for the user whose name is specified by the **user** parameter. The **basegid** parameter is usually the group number from the **/etc/passwd** file and it is automatically included in the group list.

Warning: The **initgroups** subroutine uses the **getgrent** subroutine family. If the program that invokes **initgroups** uses any of these subroutines, then calling **initgroups** overwrites the static group structure.

Return Value
Upon successful completion, the **initgroups** subroutine returns a value of 0. If the effective user ID of the calling process is not superuser, then **initgroups** returns a value of 1.

File

/etc/group

Related Information
In this book: "getgrent, getgrgid, getgrnam, setgrent, endgrent" in topic 1.2.96, "getgroups" in topic 1.2.97, and "setgroups" in topic 1.2.249.

The **adduser** command in *AIX Operating System Commands Reference*.

1.2.136 insque, remque

Purpose
Inserts or removes an element in a queue.

Library
Standard C Library **(libc.a)**

Syntax

```
struct qelem {
    struct  qelem *q_forw;
    struct  qelem *q_back;
    char q_data [ ];
};

int insque (elem, pred)
struct qelem *elem, *pred;

int remque (elem)
struct qelem *elem;
```

Description
The **insque** and **remque** subroutines manipulate queues built from doubly linked lists. Each element in the queue must be in the form of a **qelem** structure. The **q_forw** and **q_back** elements of that structure must point to the elements in the queue immediately before and after the element to be inserted or deleted.

The **insque** subroutine inserts the element pointed to by the **elem** parameter into a queue immediately after the element pointed to by the **pred** parameter.

The **remque** subroutine removes the element defined by the **elem** parameter from a queue.

1.2.137 ioctlx, ioctl, gtty, stty

Purpose
Controls input/output devices.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/ioctl.h>
#include <sys/devinfo.h>
#include <sgtty.h>


int ioctlx (fildes, op, arg,    inttgtty (fildes, argp)
int fildes, op;                 int fildes
char *arg;                      struct sgttyb *argp;
int ext;
                                int stty (fildes, argp)
int ioctl (fildes, op, arg)     int fildes
int fildes, op;                 struct sgttyb *argp;
char *arg;
```

Description
The **ioctlx** and **ioctl** system calls perform a variety of control operations on the block or character special file (device) specified by the **fildes** parameter. The **op** parameter specifies the operation, and the use of the **arg** parameter depends on the particular operation performed. The **ext** parameter provides additional device-specific information. The **ioctlx** and **ioctl** operations that are valid for each type of device are explained in Chapter 5, "Special Files." For more information on the **ioctl** and **ioctlx** operations for 4.3BSD functions, see "BSD4.3 library" in topic 1.2.22.

The following two standard calls, however, apply to any open file:

```
ioctl(fildes, FIOCLEX, 0)
ioctl(fildes, FIONCLEX, 0)
```

The first of these calls causes the file to be closed automatically during a successful **exec**, **rexec**, or **run** system call; the second causes the file to remain open across these calls (see "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71 and "fcntl.h" in topic 2.4.8).

Two operations are valid for all types of devices that support **ioctl** and **ioctlx** system calls. These two operations are:

IOCTYPE Returns the device type associated with **fildes**, left shifted 8 bits. The device types are defined in the **sys/devinfo.h** header file, which is discussed in "devinfo" in topic 2.3.15.

IOCINFO Stores device information for the file specified by **fildes** into the buffer pointed to by the **arg** parameter. See "devinfo" in topic 2.3.15 for the format of the device information structure.

Some devices support additional requests. See the discussion of individual devices in Chapter 5, "Special Files" for details about device-dependent **ioctl** calls.

Subtopics
1.2.137.1 Compatibility Interface

1.2.137.1 Compatibility Interface

ioctl (**fildes, cmd, arg**)

is equivalent to:

octlx (**fildes, cmd, arg, 0**)

The functions **gtty** and **stty** are equivalent to

```
   ioctl(fildes, TIOCGETP, argp)
  and
   ioctl(fildes, TIOCSETP, argp)
```

respectively (see "termio" in topic 2.5.28).

Return Value
If the **ioctlx**, **ioctl**, **gtty**, or **stty** call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **ioctlx**, **ioctl**, **gtty**, or **stty** calls fail if one or more of the following are true:

EBADF **fildes** is not a valid open file descriptor.

ENOTTY **fildes** is not associated with a character special file.

ENODEV The device associated with **fildes** does not support the **ioctlx** system call.

EFAULT The **arg** parameter points to a location outside of the process's allocated address space.

EINVAL **op** or **arg** is not valid.

EINTR A signal was caught during the **ioctlx** system call.

If the Transparent Computing Facility is installed on your system, **ioctlx** can also fail if one or more of the following are true:

ESITEDN1 The device is on a site which is not currently on the network.

ESITEDN2 The operation was terminated because a site failed.

Related Information
In this book: "getsockopt, setsockopt" in topic 1.2.121, "devinfo" in topic 2.3.15, and Chapter 5, "Special Files."

The **stty** command in *AIX Operating System Commands Reference*.

The discussion of **termio** in *AIX Programming Tools and Interfaces*.

1.2.138 kill, kill3, killpg

Purpose
Sends a signal to a process or to a group of processes.

Library
Standard C Library (**libc.a**)

Syntax

```
int kill (pid, sig)
pid_t pid;
int sig;

int kill3 (pid, sig, arg)
pid_t pid;
int sig, arg;
```

Description
The **kill** system call sends the signal specified by the **sig** parameter to the process or group of processes specified by the **pid** parameter. (For information on valid signals, see "sigaction, sigvec, signal" in topic 1.2.263.) If the **sig** parameter is 0 (the ***null signal***), error checking is performed but no signal is sent. This can be used to check the validity of **pid**.

To send a signal to another process, at least one of the following must be true:

Either the real or the effective user ID of the sending proces matches the real user ID or the saved set-user-ID of the receiving process.
The effective user ID of the sending process is superuser

Note: An exception to the above is the signal **SIGCONT**, which also may be sent to any process which is a descendant of the sending process. This allows a command interpreter such as **csh** to restart processes stopped by a **stop** signal sent from the keyboard, when those processes may have different real and effective user IDs.

The processes that have the process IDs 0 and 1 are special processes and are sometimes referred to here as **proc0** and **proc1**, respectively.

If the **pid** parameter is greater than 0, the signal specified by the **sig** parameter is sent to the process whose process ID is equal to the value of the **pid** parameter.

If the **pid** parameter is equal to 0, the signal specified by the **sig** parameter is sent to all of the processes, excluding **proc0** and **proc1**, whose process group ID is equal to the process group ID of the sender.

If the **pid** parameter is equal to -1, the signal specified by the **sig** parameter is sent to all of the processes, excluding system processes.

If the **pid** parameter is negative but not -1, the signal specified by the **sig** parameter is sent to all of the processes whose process group ID is equal to the absolute value of the **pid** parameter.

If the Transparent Computing Facility is installed, **kill3** may be used to send additional information with a signal. Currently, **kill3** is only

viable with **SIGMIGRATE**, in which case the **arg** parameter is the number of the site to which the processes receiving the signal will migrate. The user signal-catching routines do not receive the third argument of a **kill3** for signals other than **SIGMIGRATE**. If **arg** is 0, the processes will migrate to the sender's site. Note that **SIGMIGRATE** is only a request to migrate. If the receiving process is ignoring or catching **SIGMIGRATE**, or if the migration fails, the **kill** call succeeds but the signal does not cause the process to move. The receiving process also may not migrate immediately if it is stopped or if it is waiting for a file lock. In any case, the **kill** call usually returns before any migration has completed. The actual site on which a process is executing may be determined by using the **site** system call.

Compatibility Interfaces

The following additional interface is provided in Berkeley Compatibility Library (**libbsd.a**):

```
killpg (pgrp, sig)
int pgrp;
int sig;
```

is equivalent to:

```
if (pgrp < 0)
{
    errno = ESRCH;
    return (-1);
}
return (kill (-pgrp, sig));
```

Return Value

Upon successful completion, **kill** returns a value of 0. If **kill** fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions

The **kill** system call fails and no signal is sent if one or more of the following are true:

EINVAL **sig** is not a valid signal number.

EINVAL **sig** is **SIGKILL** and **pid** is 1 (**proc1**).

ESRCH No process can be found corresponding to that specified by **pid**.

EPERM The user ID of the sending process is not superuser, and the real or effective user ID does not match the real or saved set-user-ID of the receiving process.

If the Transparent Computing Facility is installed on your system, **kill3** can also fail if the following is true:

EPERM The user ID of the sending process is not superuser, **sig** is **SIGMIGRATE**, and the sending process does not have permission to execute on the specified site (see "getxperm, setxperm" in topic 1.2.128).

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, "getpid, getpgrp, getppid" in topic 1.2.110, "getxperm,

setxperm" in topic 1.2.128, "migrate" in topic 1.2.167, "setpgid, setpgrp, setsid" in topic 1.2.252, and "sigaction, sigvec, signal" in topic 1.2.263.

The **csh**, **kill**, and **sh** commands in *AIX Operating System Commands Reference*.

1.2.139 l3tol, ltol3

Purpose
Converts between 3-byte integers and long integers.

Library
Standard C Library (**libc.a**)

Syntax

```
void l3tol (lp, cp, n)          void ltol3 (cp, lp, n)
long *lp;                       char *cp;
char *cp;                       long *lp;
int n;                          int n;
```

Description
The **l3tol** subroutine converts a list of **n** 3-byte integers packed into a character string pointed to by the **cp** parameter into a list of long integers pointed to by the **lp** parameter.

The **ltol3** subroutine performs the reverse conversion, from long integers (**lp**) to 3-byte integers (**cp**).

Note: These routines, which in the past were useful for file system maintenance, are no longer needed as the AIX PS/2 and AIX/370 file systems use 4-byte block numbers.

Warning: The numerical values of the long integers are machine-dependent because of possible differences in byte ordering.

Related Information
In this book: "fs" in topic 2.3.20.

1.2.140 labs

Purpose
Returns the absolute value of long integers.

Library
Standard C Library (**libc.a**)

Syntax

```
long labs (n);
long n;
```

Description
The **labs** function produces the absolute value of its long integer argument **n**. There is no error-return value. The result is undefined when the argument is the least of the negative long integers (for example, -2147483648 on OS/2), whose absolute value cannot be represented as a long integer. The value of the minimum allowable integer is stored in **LONG_MIN** in the **limits.h** include file.

Example
This example computes **y** as the absolute value of the long integer -41567.

```
#include <stdlib.h>

long x, y;

x = -41567L;
y = labs (x);   /* y = 41567L */
```

1.2.141 ldahread

Purpose
Reads the archive header of a member of an archive file.

Library
Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ar.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldahread (ldptr, arhead)
LDFILE *ldptr;
ARCHDR *arhead;
```

Description
If **TYPE(ldptr**) is the archive file magic number, **ldahread** reads the archive header of the common object file currently associated with **ldptr** into the area of memory beginning at **arhead**.

The **ldahread** subroutine returns SUCCESS or FAILURE. The **ldahread** subroutine fails if **TYPE(ldptr**) does not represent an archive file, or if it cannot read the archive header.

Related Information
In this book: "ldclose, ldaclose" in topic 1.2.142, "ldopen, ldaopen" in topic 1.2.149, "ldfcn" in topic 1.2.143, and "ar" in topic 2.3.4.

1.2.142 ldclose, ldaclose

Purpose
Closes a common object file.

Library
Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>
#include <ar.h>

int ldclose (ldptr)
LDFILE *ldptr;

int ldaclose (ldptr)
LDFILE *ldptr;
```

Description
The **ldopen** and **ldclose** subroutines are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus, an archive of common object files can be processed as if it were a series of simple common object files.

If **TYPE(ldptr**) does not represent an archive file, **ldclose** closes the file and frees the memory allocated to the **LDFILE** structure associated with **ldptr**. If **TYPE(ldptr**) is the magic number of an archive file, and if there are any more files in the archive, **ldclose** reinitializes **OFFSET(ldptr**) to the file address of the next archive member and returns **FAILURE**. The **LDFILE** structure is prepared for a subsequent **ldopen**. In all other cases, **ldclose** returns **SUCCESS**.

The **ldaclose** subroutine closes the file and frees the memory allocated to the **LDFILE** structure associated with **ldptr** regardless of the value of **TYPE(ldptr**). The function is often used in conjunction with **ldaopen**.

If **ldaclose** cannot find the **LDFILE** structure associated with *ldptr*, then it returns **FAILURE**. In all other cases, **ldaclose** returns **SUCCESS**.

Related Information
In this book: "ldfcn" in topic 1.2.143 and "ldopen, ldaopen" in topic 1.2.149.

1.2.143 ldfcn

Purpose
Common object file access routines.

Library
Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>
```

Description
The common object file access routines are a collection of functions for reading common object files and archives containing common object files. Although the calling program must know the detailed structure of the parts of the object file that it processes, the routines effectively insulate the calling program from knowledge of the overall structure of the object file.

The interface between the calling program and the object file access routines is based on the defined type **LDFILE**, which is defined as **struct ldfile** and declared in the header file **ldfcn.h**. The primary purpose of this structure is to provide uniform access to both simple object files that are members of an archive file.

The function **ldopen** allocates and initializes the **LDFILE** structure and returns a pointer to the structure to the calling program. The fields of the **LDFILE** structure may be accessed individually through macros defined in **ldfcn.h** and contain the following information:

LDFILE *ldptr;

TYPE(ldptr) The file magic number used to distinguish between archive members and simple object files.

IOPTR(ldptr) The file pointer returned by **fopen** and used by the standard input/output functions.

OFFSET(ldptr) The file address of the beginning of the object file; the offset is nonzero if the object file is a member of an archive file.

HEADER(ldptr) The file header structure of the object file.

The object file access functions themselves may be divided into four categories:

Functions that open or close an object file

ldopen and **ldaopen**
open a common object file
ldclose and **ldaclose**
close a common object file

Functions that read header or symbol table information

ldahread
reads the archive header of a member of an archive file
ldfhread
reads the file header of a common object file
ldshread and **ldnshread**
read a section header of a common object file
ldtbread
reads a symbol table entry of a common object file
ldgetname
retrieves a symbol name from a symbol table entry or from the string table

Functions that position an object file at the start of the section relocation, or line number information for a particular section of a common object file:

ldohseek
seeks to the optional file header of a common object file
ldsseek and **ldnsseek**
seek to a section of a common object file
ldrseek and **ldnrseek**
seek to the relocation information for a section of a common object file
ldlseek and **ldnlseek**
seek to the line number information for a section of a common object file
ldtbseek
seek to the symbol table of a common object file

The function **ldtbindex**, which returns the index of a particular common object file symbol table entry.

All the functions except **ldopen**, **ldgetname**, **ldaopen**, and **ldtbindex** return either SUCCESS or FAILURE, constants which are defined in **ldfcn.h**. The **ldopen** and **ldaopen** functions both return pointers to an **LDFILE** structure.

Additional access to an object file is provided through a set of macros defined in **ldfcn.h**. These macros parallel the standard input/output file reading and manipulating functions, translating a reference of the **LDFILE** structure into a reference to its file descriptor field.

The following macros are provided:

GETC(ldptr)

FGETC(ldptr)

GETW(ldptr)

UNGETC(c,ldptr)

FGETS(s,n,ldptr)

FREAD((**char** *) **ptr, sizeof (*ptr), nitems,ldptr**)

FSEEK(ldptr, offset, ptrname)

FWRITE((**char** *) **ptr, sizeof (*ptr), nitems,ldptr**)

FTELL(ldptr)

REWIND(ldptr)

FEOF(ldptr)

FERROR(ldptr)

FILENO(ldptr)

SETBUF(ldptr, buf)

STROFFSET(ldptr)

The **STROFFSET** macro calculates the address of the string table in a COFF-format object file.

Warning: The macro **FSEEK**, defined in the header file **ldfcn.h**, translates into a call to the standard input/output function **fseek**. **FSEEK** should not be used to seek from the end of an archive file since the end of an archive file may not be the same as the end of one of its object file members.

Related Information
In this book: "fseek, rewind, ftell" in topic 1.2.86, "ldahread" in topic 1.2.141, "ldclose, ldaclose" in topic 1.2.142, "ldfhread" in topic 1.2.144, "ldgetname" in topic 1.2.145, "ldlread, ldlinit, ldlitem" in topic 1.2.146, "ldlseek, ldnlseek" in topic 1.2.147, "ldohseek" in topic 1.2.148, "ldopen, ldaopen" in topic 1.2.149, "ldrseek, ldnrseek" in topic 1.2.150, "ldshread, ldnshread" in topic 1.2.151, "ldtbindex" in topic 1.2.153, "ldtbread" in topic 1.2.154, and "ldtbseek" in topic 1.2.155.

1.2.144 ldfhread

Purpose
Reads the file header of a common object file.

Library
Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldfhread (ldptr, filehead)
LDFILE *ldptr;
FILHDR *filehead;
```

Description
The **ldfhread** subroutine reads the file header of the common object file currently associated with **ldptr** into the area of memory beginning at **filehead**.

The **ldfhread** subroutine returns SUCCESS or FAILURE. The **ldfhread** subroutine fails if it cannot read the file header.

In most cases, the use of **ldfhread** can be avoided by using the macro **HEADER(ldptr)** defined in **ldfcn.h** (see "ldfcn" in topic 1.2.143). The information in any field, **fieldname**, of the file header may be accessed using **HEADER(ldptr).fieldname**.

Related Information
In this book: "ldclose, ldaclose" in topic 1.2.142, "ldfcn" in topic 1.2.143, and "ldopen, ldaopen" in topic 1.2.149.

1.2.145 ldgetname

Purpose
Retrieves the symbol name for a common object file symbol table entry.

Library
Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

char *ldgetname (ldptr, symbol)
LDFILE *ldptr;
SYMENT *symbol;
```

Description
The **ldgetname** subroutine returns a pointer to the name associated with **symbol** as a string. The string is contained in a static buffer local to **ldgetname** that is overwritten by each call to **ldgetname**, and therefore, must be copied by the caller if the name is to be saved.

The common object file format handles arbitrary length symbol names with the addition of a string table. The **ldgetname** subroutine returns the symbol name associated with a symbol table entry for either a pre-COFF-format object file or a COFF-format object file. Thus, **ldgetname** can be used to retrieve names from object files without any backward compatibility problems. The **ldgetname** subroutine returns NULL (which is defined in the file **stdio.h**) for a COFF-format object file, if the name cannot be retrieved. This situation can occur:

If the string table cannot be foun

If not enough memory can be allocated for the string tabl

If the string table appears not to be a string table (for example i an auxiliary entry is handed to **ldgetname** that looks like a reference to a name in a non-existent string table)

If the name's offset into the string table is past the end of th string table.

Typically, **ldgetname** is called immediately after a successful call to **ldtbread** to retrieve the name associated with the symbol table entry filled by **ldtbread**.

Related Information
In this book: "ldclose, ldaclose" in topic 1.2.142, "ldfcn" in topic 1.2.143, "ldopen, ldaopen" in topic 1.2.149, "ldtbread" in topic 1.2.154, and "ldtbseek" in topic 1.2.155.

1.2.146 ldlread, ldlinit, ldlitem

Purpose
Manipulates line number entries of a common object file function.

Library
Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <linenum.h>
#include <ldfcn.h>
#include <scnhdr.h>
#include <syms.h>

int ldlread (ldptr, fcnindx, linenum, linent)
LDFILE *ldptr;
long fcnindx;
unsigned short linenum;
LINENO *linent;

int ldlinit (ldptr, fcnindx)
LDFILE *ldptr;
long fcnindx;

int ldlitem (ldptr, linenum, linent)
LDFILE *ldptr;
unsigned short linenum;
LINENO *linent;
```

Description
The **ldlread** subroutine searches the line number entries of the common object file currently associated with **ldptr**. This subroutine begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by **fcnindx**, the index of its entry in the object file symbol table. The **ldlread** subroutine reads the entry with the smallest line number equal to or greater than **linenum** into the memory beginning at **linent**.

The **ldlinit** and **ldlitem** subroutines together perform exactly the same function as **ldlread**. After an initial call to **ldlread** or **ldlinit**, **ldlitem** may be used to retrieve a series of line number entries associated with a single function. The **ldlinit** subroutine simply locates the line number entries for the function identified by **fcnindx**. The **ldlitem** subroutine finds and reads the entry with the smallest line number equal to or greater than **linenum** into the memory beginning at **linent**.

The **ldlread, ldlinit,** and **ldlitem** subroutines each return either SUCCESS or FAILURE. The **ldlread** subroutine fails if there are no line number entries in the object file, if **fcnindx** does not index a function entry in the symbol table, or if it finds no line number equal to or greater than **linenum**. The **ldlinit** subroutine fails if there are no line number entries in the object file or if **fcnindx** does not index a function entry in the symbol table. The **ldlitem** subroutine fails if it finds no line number equal to or greater than **linenum**.

Related Information

In this book: "ldclose, ldaclose" in topic 1.2.142, "ldfcn" in topic 1.2.143, "ldopen, ldaopen" in topic 1.2.149, and "ldtbindex" in topic 1.2.153.

1.2.147 ldlseek, ldnlseek

Purpose
Seeks to line number entries of a section of a common object file.

Library
Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>
#include <scnhdr.h>

int ldlseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnlseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

Description
The **ldlseek** subroutine seeks to the line number entries of the section specified by **sectindx** of the common object file currently associated with **ldptr**.

The **ldnlseek** subroutine seeks to the line number entries of the section specified by **sectname**.

The **ldlseek** and **ldnlseek** subroutines return SUCCESS or FAILURE. The **ldlseek** routine fails if **sectindx** is greater than the number of sections in the object file. The **ldnlseek** routine fails if there is no section name corresponding with ***sectname**. Either function fails if the specified section has no line number entries or if it cannot seek to the specified line number entries.

Note that the first section has an index of 1.

Related Information
In this book: "ldclose, ldaclose" in topic 1.2.142, "ldfcn" in topic 1.2.143, "ldopen, ldaopen" in topic 1.2.149, and "ldshread, ldnshread" in topic 1.2.151.

1.2.148 ldohseek

Purpose
Seeks to the optional file header of a common object file.

Library
Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldohseek (ldptr)
LDFILE *ldptr;
```

Description
The **ldohseek** subroutine seeks to the optional file header of the common object file currently associated with **ldptr**.

The **ldohseek** subroutine returns SUCCESS or FAILURE. The **ldohseek** routine fails if the object file has no optional header or if it cannot seek to the optional header.

Related Information
In this book: "ldclose, ldaclose" in topic 1.2.142, "ldfcn" in topic 1.2.143, "ldopen, ldaopen" in topic 1.2.149, and "ldfhread" in topic 1.2.144.

1.2.149 ldopen, ldaopen

Purpose
Opens a common object file for reading.

Library
Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>
#include <ar.h>

LDFILE *ldopen (filename, ldptr)
char *filename;
LDFILE *ldptr;

LDFILE *ldaopen (filename, oldptr)
char *filename;
LDFILE *oldptr;
```

Description
The **ldopen** and **ldclose** subroutines are designed to provide uniform access to both simple object files and object files that are members of archive files. Thus, an archive of common object files can be processed as if it were a series of simple common object files.

If **ldptr** has the value NULL, **ldopen** opens **filename**, allocates and initializes the **LDFILE** structure, and returns a pointer to the structure to the calling program.

If **ldptr** is valid and if **TYPE(ldptr**) is the archive magic number, **ldopen** reinitializes the **LDFILE** structure for the next archive member of **filename**.

The **ldopen** and **ldclose** subroutines are designed to work in concert. The **ldclose** subroutine returns FAILURE only when **TYPE(ldptr**) is the archive magic number and there is another file in the archive to be processed. Only then should **ldopen** be called with the current value of **ldptr**. In all other cases, in particular whenever a new **filename** is opened, **ldopen** should be called with a NULL **ldptr** argument.

The following example illustrates the use of the **ldopen** and **ldclose** subroutines:

```
/* for each filename to be processed */

ldptr = NULL;
do
{
     if ( (ldptr = ldopen(filename, ldptr)) != NULL )
     {
              /* check magic number */
              /* process the file */
     }
} while (ldclose(ldptr) == FAILURE );
```

If the value of **oldptr** is not NULL, **ldaopen** opens **filename** again, and

allocates and initializes a new **LDFILE** structure, copying the **TYPE, OFFSET,** and **HEADER** fields from **oldptr**. The **ldaopen** subroutine returns a pointer to the new **LDFILE** structure. This new pointer is independent of the old pointer, **oldptr**. The two pointers may be used concurrently to read separate parts of the object file. For example, one pointer may be used to step sequentially through the relocation information, while the other is used to read indexed symbol table entries.

Both **ldopen** and **ldaopen** open **filename** for reading. Both functions return NULL if **filename** cannot be opened, or if memory for the **LDFILE** structure cannot be allocated. A successful open does not insure that the given file is a common object file or an archived object file.

Related Information
In this book: "fopen, freopen, fdopen" in topic 1.2.82, "ldclose, ldaclose" in topic 1.2.142, and "ldfcn" in topic 1.2.143.

1.2.150 ldrseek, ldnrseek

Purpose
Seeks to relocation entries of a section of a common object file.

Library
Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>
#include <scnhdr.h>

int ldrseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnrseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

Description
The **ldrseek** subroutine seeks to the relocation entries of the section specified by **sectindx** of the common object file currently associated with **ldptr**. Note that the first section has an index of 1. The **ldnrseek** subroutine seeks to the relocation entries of the section specified by **sectname**.

The **ldrseek** and **ldnrseek** subroutines return SUCCESS or FAILURE. The **ldrseek** subroutine fails if **sectindx** is greater than the number of sections in the object file. The **ldnrseek** subroutine fails if there is no section name corresponding with ***sectname**. Either function fails if the specified section has no relocation entries or if it cannot seek to the specified relocation entries.

Related Information
In this book: "ldclose, ldaclose" in topic 1.2.142, "ldfcn" in topic 1.2.143, "ldopen, ldaopen" in topic 1.2.149, and "ldshread, ldnshread" in topic 1.2.151.

1.2.151 ldshread, ldnshread

Purpose
Reads an indexed or named section header of a common object file.

Library
Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <scnhdr.h>
#include <ldfcn.h>

int ldshread (ldptr, sectindx, secthead)
LDFILE *ldptr;
unsigned short sectindx;
SCNHDR *secthead;

int ldnshread (ldptr, sectname, secthead)
LDFILE *ldptr;
char *sectname;
SCNHDR *secthead;
```

Description
The **ldshread** subroutine reads the section header specified by **sectindx** of the common object file currently associated with **ldptr** into the area of memory beginning at **secthead**. Note that the first section has an index of 1. The **ldnshread** subroutine reads the section header specified by **sectname** into the area of memory beginning at **secthead**.

The **ldshread** and **ldnshread** subroutines return SUCCESS or FAILURE. The **ldshread** subroutine fails if **sectindx** is greater than the number of sections in the object file. The **ldnshread** subroutine fails if there is no section name corresponding with **sectname**. Either function fails if it cannot read the specified section header.

Related Information
In this book: "ldclose, ldaclose" in topic 1.2.142, "ldfcn" in topic 1.2.143, and "ldopen, ldaopen" in topic 1.2.149.

1.2.152 ldsseek, ldnsseek

Purpose
Seeks to an indexed or named section of a common object file.

Library
Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>
#include <scnhdr.h>

int ldsseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnsseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

Description
The **ldsseek** subroutine seeks to the section specified by **sectindx** of the common object file currently associated with **ldptr**. Note that the first section has an index of 1. The **ldnsseek** subroutine seeks to the section specified by **sectname**.

The **ldsseek** and **ldnsseek** subroutines return SUCCESS or FAILURE. The **ldsseek** subroutine fails if **sectindx** is greater than the number of sections in the object file. The **ldnsseek** subroutine fails if there is no section name corresponding with ***sectname**. Either function fails if there is no section data for the specified section or if it cannot seek to the specified section.

Related Information
In this book: "ldclose, ldaclose" in topic 1.2.142, "ldfcn" in topic 1.2.143, "ldopen, ldaopen" in topic 1.2.149, and "ldshread, ldnshread" in topic 1.2.151.

1.2.153 ldtbindex

Purpose
Computes the index of a symbol table entry of a common object file.

Library
Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

long ldtbindex  (ldptr)
LDFILE *ldptr;
```

Description
The **ldtbindex** subroutine returns the (**long**) index of the symbol table entry at the current position of the common object file associated with **ldptr**. Note that the first symbol in the symbol table has an index of 0.

The index returned by **ldtbindex** may be used in subsequent calls to the **ldtbread** subroutine. However, since **ldtbindex** returns the index of the symbol table entry that begins at the current position of the object file, if **ldtbindex** is called immediately after a particular symbol table entry has been read, it returns the index of the next entry.

The **ldtbindex** subroutine fails if there are no symbols in the object file, or if the object file is not positioned at the beginning of a symbol table entry.

Related Information
In this book: "ldclose, ldaclose" in topic 1.2.142, "ldfcn" in topic 1.2.143, "ldopen, ldaopen" in topic 1.2.149, "ldtbread" in topic 1.2.154, and "ldtbseek" in topic 1.2.155.

1.2.154 ldtbread

Purpose
Reads an indexed symbol table entry of a common object file.

Library
Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbread (ldptr, symindex, symbol)
LDFILE *ldptr;
long symindex;
SYMENT *symbol;
```

Description
The **ldtbread** subroutine reads the symbol table entry specified by **symindex** of the common object file currently associated with **ldptr** into the area of memory beginning at **symbol**. Note that the first symbol in the symbol table has an index of 0.

The **ldtbread** subroutine returns SUCCESS or FAILURE. The **ldtbread** subroutine fails if **symindex** is greater than or equal to the number of symbols in the object file, or if it cannot read the specified symbol table entry.

Related Information
In this book: "ldclose, ldaclose" in topic 1.2.142, "ldfcn" in topic 1.2.143, "ldgetname" in topic 1.2.145, "ldopen, ldaopen" in topic 1.2.149, and "ldtbseek" in topic 1.2.155.

1.2.155 ldtbseek

Purpose
Seeks to the symbol table of a common object file.

Library
Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <filehdr.h>
#include <ldfcn.h>

int ldtbseek (ldptr)
LDFILE *ldptr;
```

Description
The **ldtbseek** subroutine seeks to the symbol table of the common object file currently associated with **ldptr**.

The **ldtbseek** subroutine returns SUCCESS or FAILURE. The **ldtbseek** subroutine fails if the symbol table has been stripped from the object file, or if it cannot seek to the symbol table.

Related Information
In this book: "ldclose, ldaclose" in topic 1.2.142, "ldfcn" in topic 1.2.143, "ldopen, ldaopen" in topic 1.2.149, and "ldtbread" in topic 1.2.154.

1.2.156 link

Purpose
Creates an additional directory entry for an existing file.

Library
Standard C Library (**libc.a**)

Syntax

```
int link (path1, path2)
char *path1, *path2;
```

Description
The **link** system call creates an additional link (directory entry) for an existing file. The **path1** parameter points to the path name of an existing file and provides the inumber to be used. The **path2** parameter points to the path name for the new directory entry to be created.

If the **path1** parameter is a symbolic link, it is followed and the link is made to the file pointed at by the symbolic link. The **path2** parameter may not name an existing symbolic link.

The **path2** parameter cannot name a hidden directory; new links can only be explicitly named hidden directory components. If the **path2** parameter is an explicitly named hidden directory component, the **path1** parameter cannot name a hidden directory. If the **path1** parameter references a hidden directory, the current **xvers** string and the current site path are used to determine the file, if any, to which the link is made.

Return Value
Upon successful completion, **link** returns a value of 0. If **link** fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **link** system call fails if one or more of the following are true:

ENOTDIR A component of either path prefix is not a directory.

ENOENT A component of either path prefix does not exist.

EACCES A component of either path prefix denies search permission.

ENOENT The file named by the **path1** parameter does not exist.

EEXIST The link named by the **path2** parameter already exists.

EPERM The file named by the **path1** parameter is a directory and the effective user ID is not superuser.

EXDEV The link named by the **path2** parameter and the file named by the **path1** parameter are on different file systems.

ENOENT The **path2** parameter points to a null path name.

EACCES The requested link requires writing in a directory with a mode that denies write permission.

EROFS The requested link requires writing in a directory on a read-only file system.

EFAULT The **path1** or **path2** parameter points to a location outside of the process's allocated address space.

EDQUOT The directory in which the entry for the new link is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

EMLINK The file already has the maximum number of links.

ESTALE The process's root or current directory is located in an NFS virtual file system that has been unmounted.

ENAMETOOLONG
A component of the **path1** or **path2** parameter exceeded **NAME_MAX** characters or the entire path parameter exceeded **PATH_MAX** characters.

EACCES The **path1** parameter names a directory and **path2** names a component of a hidden directory.

ENOENT A hidden directory was named by the **path1** parameter, but no component inside it matched the process's current site path list.

ENOENT A symbolic link was named by the **path1** parameter or the path prefix of the **path2** parameter, but the file to which it refers does not exist.

ELOOP A loop of symbolic links was detected.

ENOSPC The file system is out of inodes.

ENFILE The system inode table is full.

If the Transparent Computing Facility is installed on your system, **link** can also fail if one or more of the following are true:

ESITEDN1 **path1** or **path2** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE A component of **path1** or **path2** is replicated but not stored on any site which is currently up.

EROFS Write access is requested for a directory on a replicated file system in which the primary copy is unavailable.

EINTR A signal was caught during the **link** system call.

Related Information
In this book: "unlink, rmslink, remove" in topic 1.2.318.

The **link** command in *AIX Operating System Commands Reference*.

1.2.157 listen

Purpose
Listens for connections on a socket.

Library
Internet Library (**libc.a**)

Syntax

int listen (s, backlog)
int s, backlog;

Description
To accept connections, create a socket with **socket**, specify a backlog for incoming connections with **listen**, and accept the connections with **accept**. The **listen** system call applies only to sockets of type **SOCK_STREAM**.

The **backlog** parameter defines the maximum length for the queue of pending connections. The maximum value of the **backlog** parameter is 5. If a connection request arrives with the queue full, the client may receive an error with an indication of ECONNREFUSED, or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

Return Value

Upon successful completion, a value of 0 is returned. If the **listen** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The system call fails if one or more of the following are true:

EBADF The **s** parameter is not valid.

ENOTSOCK The **s** parameter refers to a file, not a socket.

EOPNOTSUPP The referenced socket is not of type that supports **listen**.

EINVAL The referenced socket refers to a NULL inode.

Related Information
In this book: "accept" in topic 1.2.9, "connect" in topic 1.2.49, and "socket" in topic 1.2.275.

1.2.158 localeconv

Purpose
Sets components of an object type **struct lconv** with appropriate formatting values.

Syntax

#include <locale.h>

struct lconv *localeconv (int category, const char *locale);

Description

The **localeconv** subroutine sets the components of an object type **struct lconv** with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to rules of the current locale.

The members of the structure with the type **char *** are pointers to strings, any of which (except **decimal_point**) can point to **""**, to indicate that the value is not available in the current locale or is of zero length. The members with type **char** are non-negative numbers, any of which can be **CHAR_MAX,** to indicate that the value is not available in current locale. The members include the following:

Variable	**Description**
decimal_point	The decimal-point character used to format non-monetary quantities.
thousands_sep	The character used to separate groups of digits before the decimal-point character in formatted non-monetary quantities.
grouping	A string whose elements indicate the size of each group of digits in formatted non-monetary quantities.
int_curr_symbol	The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in the ISO 4217 Codes for the Representation of Currency and Funds. The fourth character (immediately preceding the NULL character) is the character used to separate the international currency symbol from the monetary quantity.
currency_symbol	The locale currency applicable to the current locale.
mon_decimal_point	The decimal-point used to format monetary quantities.
mon_thousands_sep	The separator for groups of digits before the decimal-point in formatted monetary quantities.

mon_grouping	A string whose elements indicate the size of each group of digits in formatted monetary quantities.
positive_sign	The strings used to indicate a non-negative-valued formatted monetary quantity.
negative_sign	The strings used to indicate a negative-valued formatted monetary quantity.
int_frac_digits	The number of fractional digits (those after the decimal-point) to be displayed in an internationally formatted monetary quantity.
frac_digits	The number of fractional digits (those after the decimal-point) to be displayed in a formatted monetary quantity.
p_cs_precedes	Set to 1 or 0 if the currency_symbol respectively precedes or succeeds the value for a non-negative formatted monetary quantity.
p_sep_by_space	Set to 1 or 0 if the currency_symbol is or is not separated by a space from the value for a non-negative formatted monetary quantity.
n_cs_precedes	Set to 1 or 0 if the currency_symbol respectively precedes or succeeds the value for a negative formatted monetary quantity.
n_sep_by_space	Set to 1 or 0 if the currency_symbol is or is not separated by a space from the value for a negative formatted monetary quantity.
p_sign_posn	Set to a value indicating the positioning of the positive_sign for a non-negative formatted monetary.
n_sign_posn	Set to a value indicating the positioning of the negative_sign for a negative formatted monetary quantity.

The elements of **grouping** and **mon_grouping** are interpreted according to the following

Value	**Description**
CHAR_MAX	No further grouping is to be performed.
0	The previous element is to be repeatedly used for the remainder of digits.
other	The integer value is the number of digits that comprise the current group. The next element is examined to

	determine the size of the next group of digits before the current group.

The value of **p_sign_posn** and **n_sign_posn** is interpreted according to the following.	
Valu	**Description**
0	Parentheses surround the quantity and currency_symbol.
1	The sign string precedes the quantity and currency_symbol.
2	The sign string succeeds the quantity and currency_symbol.
3	The sign strings immediately precedes the currency_symbol.
4	The sign strings immediately succeeds the currency_symbol.

The implementation behaves as if no library functions call the **localeconv** function.

Examples

The following table illustrates the rules which may be used by four countries to format monetary quantities.

Examples of monetary representations

Country	**Positive Format**	**Negative Format**	**International Format**
Italy	L.1.234	-L.1.234	ITL.1.234
Netherlan	sF 1.234,56	F -1.234,56	NLG 1.234,56
Norway	Kr1.234,56	kr11.234,56	NOK 1.234,56
Switzerla	dSFrs.1,234.	6SFrs.1,234.	6CHF 1,234.56

For these four countries, the respective values for the monetary members of the structure returned by **localeconv** are	
	Country

Variable	Italy	Netherlands	Norway	Switzerland
int_curr_symb	l"ITL."	"NLG"	"NOK"	"CHF"
currency_symb	l"L."	"F"	"kr"	"SFrs."
mon_decimal_p	i""	","	","	"."
mon_thousands	s"."	"."	"."	","
mon_grouping	"3"	"3"	"3"	"3"
positive_sign	""	""	""	""
negative_sign	"-"	"-"	"-"	"C"
int_frac_digi	s0	2	2	2
frac_digits	0	2	2	2
p_cs_precedes	1	1	1	1
p_sep_by_spac	0	1	0	0
n_cs_precedes	1	1	1	1
n_sep_by_spac	0	1	0	0
p_sign_posn	1	1	1	1
n_sign_posn	1	4	2	2

Return Value

The **localeconv** subroutine returns a pointer to the filled-in object. The structure pointed to by the return value will not be modified by the program, but may be overwritten by a subsequent call to the **localeconv** subroutine. In addition, calls to the **setlocale** subroutine with categories **LC_ALL, LC_MONETARY,** or **LC_NUMERIC** may overwrite the contents of the structure.

1.2.159 logname

Purpose
Returns the login name of the user.

Library
Programmers Workbench Library (**libPW.a**)

Syntax

char *logname ( )

Description
The **logname** subroutine returns a pointer to the null-terminated login name. The **logname** subroutine extracts the **LOGNAME** variable from the user's environment.

Note: The return value points to static data whose content is overwritten by each call. This method of determining a login name is subject to forgery. For better methods, see "cuserid" in topic 1.2.57, "getlogin" in topic 1.2.103, and "getpwent, getpwuid, getpwnam, setpwent, endpwent" in topic 1.2.114.

File
/etc/profile

Related Information
In this book: "profile" in topic 2.3.48 and "environment" in topic 2.4.6.

The **env** and **login** commands in *AIX Operating System Commands Reference*.

1.2.160 lsearch, lfind

Purpose
Performs a linear search and update.

Library
Standard C Library (**libc.a**)

Syntax

```
char *lsearch (key, base, nelp, size, compar)
char *key
char *base
unsigned int *nelp;
size_t size;
int (*compar) ( );

char *lfind (key, base, nelp, size, compar)
char *key
char *base
unsigned int *nelp;
size_t size;
int (*compar) ( );
```

Description
The **lsearch** subroutine performs a linear search generalized from Donald E. Knuth's *The Art of Computer Programming*, Volume 3, 6.1, Algorithm S. (*) It returns a pointer into a table indicating where a datum can be found. If the datum does not occur, it is added at the end of the table.

The **key** parameter points to the datum to be sought in the table. The **base** parameter points to the first element in the table. The **nelp** parameter points to an integer containing the current number of elements in the table. This integer is incremented if the datum is added to the table. The **compar** parameter is the name of the comparison function that you must supply (**strcmp**, for example). It is called with two parameters that point to the elements being compared. The **compar** function must return a value of 0 if the elements are equal and nonzero if they are not equal.

The **lfind** subroutine is identical to **lsearch**, except that if the datum is not found, then it is not added to the table. Instead, a NULL pointer is returned in this case.

The pointers to the key and the element at the base of the table should be of type pointer-to-element and cast to type pointer-to-character. Although it is declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

The comparison function need not compare every byte; therefore, the elements can contain arbitrary data in addition to the values being compared.

Warning: Undefined results can occur if there is not enough room in the table for **lsearch** to add a new item.

Example
The following code fragment reads up to **TABSIZE** strings, each of which is up to **ELSIZE** bytes long, and stores them into a table, eliminating duplicates.

```
#include <stdio.h>

#define TABSIZE 50
#define ELSIZE  120

char *lsearch();
int  strcmp();
char line[ELSIZE], tab[TABSIZE][ELSIZE];
unsigned nel = 0;
...
   while (fgets(line, ELSIZE, stdin) != NULL && nel < TABSIZE)
   {
      (void) lsearch(line, (char *)tab, &nel, ELSIZE, strcmp);
   }
...
```

Related Information

In this book: "bsearch" in topic 1.2.23, "hsearch, hcreate, hdestroy" in topic 1.2.130, and "tsearch, tdelete, twalk" in topic 1.2.309.

(*) Reading, Massachusetts: Addison-Wesley, 1981.

1.2.161 lseek

Purpose
Moves read/write file pointer.

Syntax

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek (fildes, offset, whence)
int fildes;
off_t offset;
int whence;
```

Description
The **lseek** system call sets the file pointer for the file specified by the **fildes** parameter.

The **fildes** parameter is a file descriptor obtained from a **creat, open, dup,** or **fcntl** system call.

The **lseek** system call sets the file pointer associated with the **fildes** stream according to the value of the **whence** parameter, as follows:

SEEK_SET Sets the file pointer to the value of the **offset** parameter.

SEEK_CUR Sets the file pointer to its current location plus the value of the **offset** parameter.

SEEK_END Sets the file pointer to the size of the file plus the value of the **offset** parameter.

Return Value
Upon successful completion, the resulting pointer location as measured in bytes from the beginning of the file is returned. A negative value may be returned if the **offset** parameter was negative; **errno** is not set. If **lseek** fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **lseek** system call fails and the file pointer remains unchanged if one or more of the following are true:

EBADF **fildes** is not an open file descriptor.

ESPIPE **fildes** is associated with a pipe (FIFO) or a multiplexed special file.

EINVAL **whence** is not **SEEK_SET, SEEK_CUR**, or **SEEK_END**.

If the Transparent Computing Facility is installed on your system, **lseek** can also fail if one or more of the following are true:

ESITEDN1 The file identified by **fildes** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

Related Information

In this book: "dup" in topic 1.2.64, "fcntl, flock, lockf" in topic 1.2.78, "fseek, rewind, ftell" in topic 1.2.86, and "open, openx, creat" in topic 1.2.199.

1.2.162 malloc, free, realloc, calloc, valloc, alloca, mallopt, mallinfo

Purpose
Provides a means to allocate memory.

Default malloc subroutines

Library
Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
#include <sys/types.h>

void *malloc (size)
size_t size;

void free (ptr)
void *ptr;

void *realloc (ptr, size)
void *ptr;
size_t size;

void *calloc (nelem, elsize)
size_t nelem, elsize;
```

Alternate malloc subroutines

Library
Malloc Library (**libmalloc.a**)

Syntax

```
#include <malloc.h>

char *malloc (size)
unsigned size;

void free (ptr)

char *ptr

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;

int mallopt (cmd, value)
int cmd, value;

struct mallinfo mallinfo ();
```

Description

The **malloc** and **free** subroutines provide a simple general purpose memory allocation package.

Two different **malloc** suites are provided. The default **malloc** suite includes **malloc**, **free**, **realloc**, **calloc**, **valloc**, and **alloca**. The alternate **malloc** suite includes **malloc**, **free**, **realoc**, **calloc**, **mallopt**, and **mallinfo**. To use the alternate **malloc**, add **-1malloc** when running **cc** or **ld**. Except where noted, the four subroutines shared by both suites (**malloc**, **free**, **realloc**, and **calloc**) behave the same.

The default **malloc** allocates memory in a block which is a power of two. The default **malloc** is faster than the alternate **malloc** because the allocated memory is page aligned. However, it is more wasteful of memory than the alternate **malloc**.

The alternate **malloc** adds only a small header to each piece allocated, so it doesn't use as much memory as the default **malloc**. The memory allocated

by the alternate **malloc** is not page aligned, so the **malloc** runs slower because of the increased paging.

If your program needs to use as much of the available memory as possible, you should consider using the alternate **malloc** suite, or use the **sbrk** system call directly. (Note that the alternate **malloc** doesn't *automatically* allocate as much of the available memory as possible.) This is especially relevant for AIX/370 programs compiled in S/370 (non-XA) mode. These processes have a smaller virtual address space than XA-mode processes or AIX PS/2 processes.

The **malloc** subroutine returns a pointer to a block of at least *size* bytes. The block is aligned so that it can be used for any type of data. Undefined results occur if the space assigned by **malloc** is overrun.

The **malloc** subroutine searches memory for the first contiguous area of free space of at least *size* bytes. The search is performed in a circular pattern from the last block of memory allocated or freed. During the search, **malloc** joins adjacent free blocks of memory. If a large enough contiguous area of free space is not found, then **malloc** issues an **sbrk** system call to get more memory from the system.

The **free** subroutine frees the block memory pointed to by the *ptr* parameter for further allocation. The block pointed to by the *ptr* parameter must have been previously allocated by the **malloc** subroutine. The default **free** subroutine does not change the contents of this block of memory, but the alternate **free** subroutine does, unless you use the **M_KEEP** directive of **mallopt**. Undefined results occur if the *ptr* parameter is not a valid pointer.

The **realloc** subroutine changes the size of the block of memory pointed to by the *ptr* parameter to the number of bytes specified by the *size* parameter, and then it returns a pointer to the block. The contents of the block remain unchanged up to the lesser of the old and new sizes. If a large enough block of memory is not available, then **realloc** calls the **malloc** subroutine to enlarge the memory area, and then moves the data to the new space.

The **calloc** subroutine allocates space for an array with the number of elements specified by the *nelem* parameter. Each element is of the size specified by the *elsize* parameter. The space is initialized to 0's.

Each of the allocation subroutines returns a pointer to space suitably aligned for storage of any type of object. Cast the pointer to the type *pointer-to-element* before using it.

mallopt (part of the alternate **malloc** suite) provides for control over the allocation algorithm. The available values for *cmd* are:

M_MXFAST Set *maxfast* to *value*. The algorithm allocates all blocks below the size of *maxfast* in large groups and then doles them out very quickly. The default value for *maxfast* is 24.

M_NLBLKS Set *numlblks* to *value*. The "large groups" mentioned in the previous item each contain *numlblks* blocks. *numlblks* must be greater than 0. The default value for *numlblks* is 100.

M_GRAIN Set *grain* to *value*. The sizes of all blocks smaller than *maxfast* are considered to be rounded up to the nearest multiple of *grain*. *grain* must be greater than 0. The

default values of *grain* is the smallest number of bytes which will allow alignment of any data type. Value will be rounded up to a multiple of the default when *grain* is set.

M_KEEP Preserve data in a freed block until the next **malloc**, **realloc**, or **calloc**. This option is provided only for compatibility with the default version of **malloc** and is not recommended.

These values are defined in the **<malloc.h>** header file.

mallopt may be called repeatedly, but may not be called after the first small block is allocated. If **mallopt** is called after any allocation or if *cmd* or *value* are invalid, non-zero is returned. Otherwise, it returns zero.

mallinfo (part of the alternate **malloc** suite) provides instrumentation describing space usage. It returns the structure:

```
struct mallinfo  {
     int arena;              /* total space in arena */
     int ordblks;           /* number of ordinary blocks */
     int smblks;            /* number of small blocks */
     int hblkhd;            /* space in holding block headers */
     int hblks;             /* number of holding blocks */
     int usmblks;           /* space in small blocks in use */
     int fsmblks;           /* space in free small blocks */
     int uordblks;          /* space in ordinary blocks in use */
     int fordblks;          /* space in free ordinary blocks */
     int keepcost;          /* space penalty if keep option is used */
}
```

This structure is defined in the **<malloc.h>** header file.

The **malloc**, **realloc**, and **calloc** subroutines return a NULL pointer if there is no available memory or if the memory arena has been corrupted by storing outside the bounds of a block. When this happens, the block pointed to by the *ptr* parameter could be destroyed.

Error Conditions
The **malloc**, **realloc**, and **calloc** subroutines will fail if the following is true:

ENOMEM Insufficient memory is available.

Compatibility Note

The **valloc** subroutine, found in many BSD systems, is supported as a compatibility interface. It is used by compiling the Berkeley Compatibility Library (**libbsd.a**). The function of **valloc** is superceeded by **malloc**, which automatically page aligns large (greater than or equal to 1 page) requests. Its syntax is as follows:

char *valloc (*size***)**
unsigned int *size***;**

The **alloca** subroutine is included also for compatibility with older BSD programs and is used by compiling with the Berkeley Compatibility Library (**libbsd.a**). This subroutine allocates **size** bytes of space in the stack frame of the caller. This temporary space is automatically freed on

return. Its syntax is as follows:

```
include <alloca.h>

char *alloca (size)
int size;
```

1.2.163 matherr

Purpose
Performs an action when a math subroutine encounters an error.

Library
Math Library (**libm.a**)

Syntax

```
#include <math.h>

int matherr (excp)
struct exception *excp;
```

Description
If **matherr** error handling is in effect, then the math library subroutines call **matherr** when an error is detected. See "cbrt, exp, expm1, log, log10, log1p, pow, sqrt" in topic 1.2.28 and "sin, cos, tan, asin, acos, atan, atan2" in topic 1.2.270 about alternative error handling available for these subroutines.

You can override the default error-handling actions by supplying a subroutine of your own in place of the **matherr** subroutine supplied in the math library. To do this, include in your program a subroutine named **matherr** that takes one parameter: a pointer to an **exception** structure. The **exception** structure is defined in the **math.h** header file and it contains the following members:

```
int     type;
char    *name;
double  arg1, arg2, retval;
```

The structure member named **type** describes the type of error that occurred. Its value is one of the following constants:

DOMAIN	Domain error
SING	Singularity
OVERFLOW	Overflow
UNDERFLOW	Underflow
TLOSS	Total loss of significance
PLOSS	Partial loss of significance

The **name** member points to a string containing the name of the subroutine that encountered the error. The members **arg1** and **arg2** contain the parameters that were passed to the subroutine. The **retval** member is the value that the math subroutine returns.
All of the math subroutines that call **matherr** do so in ways similar to this:

```
/*
** Set up the exception structure
*/
exc.type = DOMAIN;    /* Type of error         */
exc.name = "pow";     /* Name of subroutine    */
exc.arg1 = x;         /* Arguments to pow(x,y) */
exc.arg2 = y;

if (matherr(&exc) == 0)
```

```
  {
     /*
     ** matherr returned 0, so perform the
     ** default error-handling procedures
     */
     fprintf(stderr, "pow: DOMAIN error\n");
     exc.retval = 0;
     errno = EDOM;
  }

  return (exc.retval);
```

Studying this sample shows that the return value from the **matherr** subroutine controls whether or not the math subroutine performs its default error-handling procedures. If **matherr** returns 0, then the default procedures are performed. Note in particular that if you want to specify the value to be returned by the math subroutine, then your **matherr** subroutine must set **excp->retval** and return a nonzero value.

If you do not supply your own **matherr** subroutine, then the **matherr** subroutine supplied in the math library is linked into your program. This subroutine does nothing except return the value 0. Because it returns 0, the calling math subroutine then performs its default error-handling procedures. The default procedures are mentioned in the discussion of each math subroutine.

The math library subroutines **atan**, **ceil**, **erf**, **erfc**, **fabs**, **floor**, **fmod**, and **tanh** do not generate any of the error types listed on page 1.2.163 and therefore do not call **matherr**.
The following table shows the default error-handling procedures for the remaining math library subroutines:

Figure 2-1. Default Error-Handling Procedures

	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
acos	M,0,D(1)	--	--	--	--	--
acosh	M,H,D(10	--	--	--	--	--
asin	M,0,D(1)	--	--	--	--	--
asinh	--	--	--	--	--	--
atan2	M,0,D(2)	--	--	--	--	--
atanh	M,H,D(11	--	--	--	--	--
cabs	--	--	--	--	--	--
cbrt	--	--	--	--	--	--
cos	--	--	--	--	M,0,R	*,R
cosh	--	--	H,R	--	--	--
exp	--	--	H,R	0,R	--	--
expml	--	--	H,R	0,R	--	--

gamma	--	M,H,D(3)	H,R	--	--	--
hypot	--	--	H,R	--	--	--
j0	--	--	--	0,R	M,0,R	--
j1	--	--	--	0,R	M,0,R	--
jn	--	--	--	0,R	--	--
lgamma	M,H,D(3)	--	M,H,R	M,H,R	--	--
log	M,-H,D(4	M,-H,R(5	--	--	--	--
log10	M,-H,D(4	M,-H,R(5	--	--	--	--
loglp	M,±H,D(1	)--	--	--	--	--
pow	M,0,D(6)	--	±H,R	0,R	--	--
sin	--	--	--	--	M,0,R	*,R
sinh	--	--	±H,R	--	--	--
sqrt	M,0,D(7)	--	--	--	--	--
tan	--	--	±H,R	--	M,0,R	*,R
y0	M,-H,D(8	--	-H,R(9)	0,R	M,0,R	--
y1	M,-H,D(8	--	--	0,R	M,0,R	--
yn	M,-H,D(8	--	-H,R(9)	0,R	--	--

The following abbreviations are used in the table:

*	As much as possible of the value is returned.
0	0 is returned.
H	HUGE is returned.
-H	-HUGE is returned.
±H	HUGE or -HUGE is returned.
M	A message is written to **stdout**.
D	**errno** is set to EDOM.
R	**errno** is set to ERANGE.

Notes:

(1) Caused by passing **acos** or **asin** a value larger than 1.0.

(2) Caused by trying to calculate **atan2**(0, 0).

(3) Caused by passing **gamma** or **lgamma** a nonpositive integer.

(4) Caused by passing **log** or **log10** a negative value.

(5) Caused by trying to calculate **log**(0) or **log10**(0).

(6) Caused by trying to raise a negative number to a noninteger power or 0

to a nonpositive power.

(7) Caused by passing **sqrt** a negative value.

(8) Caused by passing **y0**, **y1**, or **yn** a nonpositive value.

(9) Caused by passing **y0** a very small positive value.

(10) Caused by passing **acosh** a value smaller than 1.0.

(11) Caused by passing **atanh** a value whose absolute value is greater than or equal to 1.0.

(12) Caused by passing **loglp** a value which is less than or equal to -1.0.

Examples

The following subroutine suggests the kinds of actions that a user-supplied **matherr** subroutine might perform. It is **not** the **matherr** subroutine that is provided in the math library. The supplied **matherr** subroutine merely returns 0.

```
int matherr(x)
register struct exception *x;
{
   switch (x->type)
   {
      case DOMAIN:
      case SING:
         /* Display message and abort */
         fprintf(stderr, "domain error in %s\n", x->name);
         abort();

      case OVERFLOW:
         if (strcmp("exp", x->name) == 0)
         {
            /* If exp, display message & return the argument */
            fprintf(stderr, "exp of %f\n", x->arg1);
            x->retval = x->arg1;
         }
         else
            if (strcmp("sinh", x->name) == 0)
            {
               /* If sinh, set errno, return 0 */
               errno = ERANGE;
               x->retval = 0;
            }
            else
               /* Otherwise, return HUGE */
               x->retval = HUGE;
         break;

      case UNDERFLOW:
         return (0); /* Perform the default procedures */


      case TLOSS:
      case PLOSS:
         /* Display message and return 0 */
         fprintf(stderr, "loss of significance in %s\n",
            x->name);
```

```
            x->retval = 0;
            break;
      } /* switch  */

      return (1); /* Do NOT perform the default procedures */
  }    /* matherr */
```

Related Information

In this book: "math.h" in topic 2.4.13.

1.2.164 mbstring

Purpose
Performs operations on strings.

Syntax

```
#include <string.h>
#include <mbcs.h>


                char *mbscat (s1, s2)       char *mbspbrk (s1, s2)
                char *s1, s2;               const char *s1, *s2;

                char *mbscpy (s1, s2)       size_t mbsspn (s1, s2)
                char *s1, s2;               const char *s1, *s2;

                size_t mbslen (s)           size_t mbscspn (s1, s2)
                char *s;                    const char *s1, *s2;

                char *mbsncat (s1, s2, ncharchar *mbstok (s1, s2)
                char *s1, *s2;              char *s1;
                size_t nchar;               const char *s2;

                char *mbsncpy (s1, s2, ncharint mbscmp (s1, s2)
                char *s1, *s2;              char *s1, *s2;
                size_t nchar;
                                            int mbsncmp (s1, s2, nchar)
                char *mbschr (s, c)         char *s1, s2;
                char *s;                    size_t nchar;
                mbchar_t c;
                                            char *mbsadvance (sn)
                char *mbsrchr (s, c)        size_t n;
                const char *s;
                mbchar_t c;

                char *mbsinvalid (s)
                const char *s;

#include <stdlib.h>

int mblen (s, n)
const char *s;
size_t n;


#include <stdlib.h>

int mbdwidth (mbs)
char *mbs;
```

Description
The **mbsncat** subroutines appends **nchar** multibyte characters of one string **s2** to another string, **s1**. This subroutine does not guarantee that the **s1** buffer will not overflow.

Similarly, the **mbsncpy** subroutine copies **nchar** multibyte characters.

The **mbschr** subroutine returns a pointer to the first occurrence of the

multibyte character **c** in the string **s**. A NULL pointer is returned if **c** does not occur in the string.

The **mbsrchr** subroutine returns a pointer to the last occurrence of the multibyte character specified by **c** in the string **s**. A NULL pointer is returned if **c** does not occur in the string, or if **c** is an invalid character or NULL. A NULL is also returned if **s** is an invalid character.

The **mbspbrk** subroutine returns a pointer to the first occurrence in the string pointed to by the **s1** parameter of any character from the string pointed to by the **s2** parameter. A NULL pointer is returned is returned if no multibyte character matches.

The **mbslen** subroutine returns the number of multibyte characters in the string pointed to by **s**.

The **mbsspn** subroutine returns the length in bytes of the initial segment of the string pointed to by the **s1** parameter, consisting entirely of characters from the string pointed to by the **s2** parameter.

The **mbscspn** subroutine returns the length in bytes of the initial segment of the string pointed to by the **s1** parameter, consisting entirely of characters NOT from the string pointed to by the **s2** parameter.

The **mbstok** subroutine returns a pointer to an occurrence of a text token in the string pointed to by the **s1** parameter. The **s2** parameter specifies a set of token delimiters. At the found character in **s1**, a null character is replaced and a pointer to the first character of the text token is returned.

The **mbscmp** subroutine compares multibyte characters in one string **s1** and the another string **s2**. The comparison is based on the binary ordering of the characters. Therefore, all two-byte characters are greater than one-byte **chars**, independent of the current locale. The return value is zero, greater than zero or less than zero.

The **mbsncmp** subroutine is the same as the **mbscmp** subroutine except that the comparison is done on **nchar** characters.

If **s** does not point to a NULL character, the **mblen** subroutine returns the number of bytes in the first multibyte character pointed to by **s**.

If **s** points to a NULL character, or to the number of bytes in a valid multibyte character (consisting of the next **n** or fewer bytes), **mblen** returns the value ZERO; if **s** points to an invalid multibyte character (consisting of greater than **n** bytes), **mblen** returns the value -1.

The return value of **mblen** is never greater than **n** or the value of **MB_CUR_MAX**, which is a code-set specific constant that represents the maximum number of bytes per character.

If **mbs** is not a NULL pointer, the **mbdwidth** subroutine determines the display width of the multibyte character pointed to by **mbs**. If **mbs** points to an invalid multibyte character (that is, a character whose value is greater than the value of **MB_CUR_MAX**), **mbdwidth** returns a value of -1; otherwise, it returns the display width of the first character. The file code used to define the multibyte character is specified by the **LC_CTYPE** or the **LC_ALL** category of the locale.

The **mbsinvalid** subroutine returns a pointer to the byte following the last

valid multibyte character (as specified by **LC_CTYPE** or **LC_ALL**) in the string pointed to by **s**. If all the characters in the string pointed to by **s** are valid multibyte characters, **mbsinvalid** returns a NULL pointer.

If **s** is not a NULL pointer, the **mbsadvance** subroutine returns a pointer to the next multibyte character (as specified by **LC_CTYPE** or **LC_ALL**) in the string pointed to by **s**, after skipping the character at the beginning of the string.

Related Information
In this book: "wcstring" in topic 1.2.327, "NLstring" in topic 1.2.193, "NCstring" in topic 1.2.184, "string" in topic 1.2.288, "stdlib.h" in topic 2.4.24, "limits.h" in topic 2.4.11, "setlocale" in topic 1.2.251, "string.h" in topic 2.4.25, and "mbcs.h" in topic 2.4.14.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.165 mbtowc, mbstowcs, mbstomb

Purpose
Converts multibyte characters and multibyte character strings into wide characters and wide character strings.

Syntax

```
#include <string.h>
#inclued <mbcs.h>

int mbtowc (pwc, s, n)
wchar_t *pwc
char *s;
size_t n;

size_t mbstowcs (wcs, s, n)
wchar_t *wcs;
char *s;
size_t n;

int mbstomb (mbch, mbs, n)
mbchar_t *mbch;
char *mbs;
size_t n;
```

Description
The **mbtowc** subroutine converts a multibyte character to a wide character, returns the number of bytes of the multibyte character and stores the result in **s**. If **pwc** is a NULL pointer, the number of bytes needed to convert **s** to a wide character is returned.

The **mbstowcs** subroutine converts the string of multibyte characters **s** to a wide character string and stores the result in **wcs**. No more than **n** wide characters are placed in the **wcs** string.

The **mbstomb** subroutine converts the first multibyte character in the string **mbs** to a character of a type **mbchar_t** and places it in **mbch** without changing its value. No more than **n** bytes can be placed in the **mbchar_t** character. The file code used to define the multibyte character is specified by the **LC_CTYPE** or the **LC_ALL** category of the locale.

Return Value
The **mbtowc** subroutine returns:

m where **m** is the number of bytes converted.

-1 if **s** does not contain a valid multibyte character.

0 if **s** is a NULL pointer.

The **mbstwcs** subroutine returns:

m where **m** is the number of wide characters converted.

-1 if **s** does not contain a valid multibyte character.

0 if **s** is a NULL pointer.

The **mbstomb** subroutine returns:

m where **m** is the number of bytes in the multibyte character.

-1 if **mbch** or **mbs** is a NULL pointer, or if the first multibyte character in **mbs** exceeds **n**.

Related Information
In this book: "wctomb, wcstombs" in topic 1.2.328 and "setlocale" in topic 1.2.251.

The **mbcsgen** command in *AIX Operating System Commands Reference*.

AIX Guide to MultiByte Character Set (MBCS) Support.

1.2.166 memory: memccpy, memchr, memcmp, memcpy, memset, bcopy

Purpose
Performs memory operations.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <string.h>
#include <memory.h>

char *memccpy (target, sourcchar,*memcpy (target, source, n)
char *target, *source;      char *target, *source;
int c;                      size_t n;
size_t n;
                            char *memset (s, c, n)
char *memchr (s, c, n)      char *s;
char *s;                    int c;
int c;                      size_t n;
size_t n;
                            void bcopy (source, target, n)
int memcmp (target, source, char *source, *target;
void *target, *source;      int n;
size_t n;
```

Description
The **memory** subroutines operate on memory areas. A memory area is an array of characters bounded by a count, and not terminated by a null character. The **memory** subroutines do not check for the overflow of any receiving memory area. All of the **memory** subroutines are declared in the **memory.h** header file.

The **memccpy** subroutine copies characters from memory area **source** into memory area **target**. The **memccpy** subroutine stops after the first character **c** is copied, or after **n** characters have been copied, whichever comes first. **memccpy** returns a pointer to the character after **c** is copied into **target**, or a NULL pointer if **c** is not found in the first **n** characters of **source**.

The **memchr** subroutine returns a pointer to the first occurrence of character **c** in the first **n** characters of memory area **s**, or a NULL pointer if **c** does not occur.

The **memcmp** subroutine lexicographically compares the first **n** characters in memory area **target** to the first **n** characters in memory area **source**. **memcmp** uses native character comparison, which may be signed on some machines. The **memcmp** subroutine returns the following values:

Less than 0	If **target** is less than **source**
Equal to 0	If **target** is equal to **source**
Greater than 0	If **target** is greater than **source**.

The **memcpy** subroutine copies **n** characters from memory area **source** to area **target** and returns **target**. This routine does not correctly handle overlapped copies.

The **memset** subroutine sets the first **n** characters in memory area **s** to the value of character **c** and returns **s**.

Like the **memcpy** subroutine, the **bcopy** subroutine copies **n** characters from memory area **source** to area **target**. However, **bcopy** handles overlaps. The order of the parameters is reversed for **bcopy**, with **source** being specified first. This subroutine has no return values.

Warning: Character movement is performed differently in different implementations of these subroutines; therefore, overlapping moves may yield unexpected results.

Related Information
In this book: "string" in topic 1.2.288 and "swab" in topic 1.2.292.

1.2.167 migrate

Purpose
Moves a process to another cluster site.

Syntax

#include <sys/types.h>

int migrate(site_number)
siteno_t site_number;

Description
The **migrate** system call moves the calling process to a specified site. If **site_number** is 0, the site is chosen using the process's site path (see "getspath, setspath" in topic 1.2.122).

If the chosen site is not the process's current site, an accounting record is written indicating the process's utilization of the old site.

The new process (on the new site) is an exact copy of the calling process (on the old site). If the new site is the same as the old site, the call succeeds but does nothing. If the new site is not the old site, the new process inherits the following attributes from the calling process:

Environmen

Open file

Close-on-exec flag (see "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71)

Signal handling settings (that is, **SIG_DFL**, **SIG_IGN**, and function address)

Signal mechanism new/old status (see "sigaction, sigvec, signal" in topic 1.2.263)

Set-user-ID and set-group-ID mode bits (see "setxuid" in topic 1.2.256)

Profiling on/off statu

Nice value (see "getpriority, setpriority, nice" in topic 1.2.111)

Process I

Parent process I

Process group I

Session I

TTY group ID (see "exit, _exit" in topic 1.2.73 and "sigaction, sigvec, signal" in topic 1.2.263)

Time left until an alarm clock signal (see "alarm" in topic 1.2.14)

Current working director

Root director

<LOCAL> alias path name (see "getlocal, setlocal" in topic 1.2.102)

File mode creation mask (see "umask" in topic 1.2.314)

File locks (see "fcntl, flock, lockf" in topic 1.2.78)

System resource limits (see "ulimit" in topic 1.2.313) and "getrlimit, setrlimit, vlimit" in topic 1.2.115)

Site path (see "getspath, setspath" in topic 1.2.122)

Execution site permissions (see "getxperm, setxperm" in topic 1.2.128).

If the new site is not the same as the old site, the new process will differ from the calling process in that the new processes **cutime**, **cstime**, **utime**, and **stime**, are modified as follows:

cutime += **utime**;

cstime += **stime**;

utime = 0;

stime = 0;

Return Value
Upon successful completion, a value of 0 is returned to the calling process. Otherwise, a value of -1 is returned, the process continues on its old site, and **errno** is set to indicate the error.

Error Conditions
The **migrate** system call will fail if:

EAGAIN The effective user ID is not superuser, and the system-imposed limit on the total number of processes under execution by a single user on the new site would be exceeded.

EBADST **site_number** is out of range or the destination site is not the same CPU type as the current site.

ESITEDN1 Operation failed because a required site is unavailable.

EPERM Execute permission is not granted for **site_number** (when not equal to 0) or for any of the sites chosen by the site path (when **site_number** is 0).

ETABLE On either the new site or the old site, the system's PID-site table, which is used to keep track of remote processes and process groups, is full.

ENLDEV The process may not execute on the designated site because one of its open file descriptors is for a local-only object such as a socket or a non-**tty** character special file. The process may not execute to another site if any TCP/IP sockets are open. See "TCP/IP Communication" in topic 1.2.2.8.

ELOCALONLY

The process may not migrate because it has made use of shared memory, semaphores, message operations (see "Semaphores, Message Queues, and Shared Memory Segments" in topic 1.2.2.10), too many child processes, or it is a DOS process (see *DOS Merge User's and Administrator's Guide*).

ENOSTORE The current load module cannot be located from site *site number* because it has been deleted or superceded by a new version.

Restrictions

Note: Processes may not execute to another site if:

1. They have too many (85 or more) child processes.
2. They have a file open which is marked as being in error (for instance, the storage site is not on the cluster network)
3. They have made use of shared memory, semaphores, or messages operations (see "Semaphores, Message Queues, and Shared Memory Segments" in topic 1.2.2.10)
4. There are any TCP/IP sockets open (see "TCP/IP Communication" in topic 1.2.2.8).
5. They have a character device open (other than a terminal or a null device)
6. They have an open descriptor for a file that has been unlinked.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, " fork, vfork" in topic 1.2.83, "getspath, setspath" in topic 1.2.122, "getxperm, setxperm" in topic 1.2.128, " rexec: rexecl, rexecv, rexecle, rexecve, rexeclp, rexecvp" in topic 1.2.236, "rfork" in topic 1.2.237, and " run: runl, runv, runle, runve, runlp, runvp" in topic 1.2.239.

1.2.168 mkdir

Purpose
Creates a directory.

Syntax

```
int mkdir (path, mode)
char *path;
int mode;
```

Description
The **mkdir** system call creates a new directory. The **path** parameter names the new directory.

To execute the **mkdir** system call, a process must have search permission to get to the parent directory of **path** and write permission in the parent directory.

The new directory has:

The owner ID set to the process's effective user ID

The group ID set to

- The group ID of its parent directory, if the parent directory has the *set-file-group-ID* attribute; otherwise,

- The process effective group ID

Permission and attribute bits set according to the value of the *mode* parameter, with the following modifications:

- All bits set in the process file mode creation mask are cleared. (For information about the file mode creation mask, see "umask" in topic 1.2.314).

- The *set-file-group-ID* and *sticky* attributes are inherited from the parent directory.

Return Value
Upon successful completion, the **mkdir** system call returns a value of 0. If the **mkdir** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **mkdir** system call fails and the directory is not created if one or more of the following are true:

ENOTDIR A component of the path is not a directory.

ENOENT A component of the path does not exist.

EACCES Creating the requested directory requires writing in a directory with a mode that denies write permission.

EACCES Search permission is denied for a component of the path.

EROFS The named file resides on a read-only file system.

EEXIST The named file already exists.

EFAULT The **path** parameter points outside of the process's allocated address space.

EIO An I/O error occurred while writing to the file system.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

ENAMETOOLONG
A component of the **path** parameter exceeded **NAME_MAX** characters or the entire path parameter exceeded **PATH_MAX** characters.

EACCES The **path** parameter explicitly named a hidden directory component.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

ELOOP A loop of symbolic links was detected.

ENOSPC The file system is out of inodes, or there is not enough space to increase the parent directory size.

EDQUOT The directory in which the entry for the new link is being placed cannot be extended because the user's quota of disk blocks or inodes on the file system containing the directory has been exhausted.

If the Transparent Computing Facility is installed on your system, **mkdir** can also fail if one or more of the following are true:

ESITEDN1 **path** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE **path** is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of **path** is replicated but is not stored on any site which is currently up.

EROFS Write access is requested for a file on a replicated file system in which the primary copy is unavailable.

EINTR A signal was caught during the system call.

Related Information
In this book: "chmod, fchmod" in topic 1.2.44, "mknod, mknodx, mkfifo" in topic 1.2.169, "rename" in topic 1.2.233, "rmdir" in topic 1.2.238, and "umask" in topic 1.2.314.

1.2.169 mknod, mknodx, mkfifo

Purpose
Creates a directory, a special file, a FIFO special file, or an ordinary file.

Syntax

```
#include <sys/stat.h>

int mknod (path, mode, dev)
char *path;
int mode;
dev_t dev;

int mknodx(path, mode, site, dev)
char *path;
int mode;
siteno_t site;
dev_t dev;
```

Description
The **mknod** system call creates a new regular file, special file, FIFO special file, or directory. The **path** parameter names the new file.

Also see "mkdir" in topic 1.2.168 for additional information on creating a directory.

The **mode** parameter specifies the ***mode*** of the file, which defines the file type and access permissions.

The **dev** parameter is configuration dependent and is used only if the **mode** parameter specifies a block or character special file. **dev** is the ID of the device, and it corresponds to the **st_rdev** member of the structure returned by the **stat** system call. See "statx, fstatx, stat, fstat, fullstat, ffullstat, lstat" in topic 1.2.282 and "stat.h" in topic 2.4.22 for more information about the device ID.

The **mode** parameter is constructed logically ORing the values specified in "chmod, fchmod" in topic 1.2.44 with one the following values, which define the file type:

S_IFDIR	Directory
S_IFCHR	Character special file
S_IFMPX	Multiplexed character special file
S_IFBLK	Block special file
S_IFREG	Regular data file
S_IFIFO	FIFO special file.

The file types **S_HIDDEN, S_SOCKET,** and **S_LINK** cannot be created using **mknod** or **mknodx**. Use **chhidden** to create **S_HIDDEN**, **bind** to create **S_SOCKET**, and **symlink** to create **S_LINK**.

A complete list of the possible **mode** values and other useful macros appears in "stat.h" in topic 2.4.22.

The new file has:

the owner ID set to the process effective user ID

the group ID set to

- the group ID of its parent directory, if the parent directory has the *set-file-group-ID* attribute; otherwise,

- the process effective group ID.

permission and attribute bits set according to the value of the *mode* parameter, modified as follows:

- All bits set in the process file mode creation mask are cleared. (For information about the file mode creation mask, see "umask" in topic 1.2.314.)

- If the new file is a directory, its *set-file-group-ID* and *sticky* attributes are inherited from the parent directory.

If the type of the new file is **S_IFMPX** (multiplexed character special file), then when the file is used, additional path name components can appear after the path name as if it were a directory. The additional part of the path name is available to the file's device driver for interpretation. This provides a multiplexed interface to the device driver. The **hft** device driver uses this feature. (See "hft" in topic 2.5.11 for details about this device driver.)

Upon successful completion, the **mknod** system call causes the **st_ctime** and **st_mtime** fields of the directory in which the new regular file, special file, FIFO special file, or directory was created to be updated to the current time. Upon successful completion, the **mknod** system call causes the **st_atime**, **st_ctime**, and **st_mtime** fields of the new regular file, special file, FIFO special file, or directory to be set to the current time. See "stat.h" in topic 2.4.22 for information on the **st_atime**, **st_ctime**, and **st_mtime** fields.

The **mknod** system call can be invoked only by the superuser for file types other than FIFO special file.

If the Transparent Computing Facility is installed, the **mknodx** system call may be used to create character or block special files for devices on any site in the cluster. The **site** parameter specifies the site which has the physical device. A **site** parameter value of 0 indicates a generic device special file; this always refers to a device on the site from which it is accessed, rather than a specific device on a specific site. Calling the **mknod** system call is equivalent to calling the **mknodx** system call with the **site** parameter as the local site number. Therefore, device special files created within a TCF cluster using **mknod** refer to devices on the local node, not devices on the node where the file resides. The **site** parameter is ignored if creating other types of files.

Subtopics
1.2.169.1 Compatibility Interfaces

1.2.169.1 Compatibility Interfaces

The following interface is provided to allow for POSIX (draft 13, section 5.4.2) conformity.

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo (path, mode)
char *path;
mode_t mode;
```

mkfifo (path, mode)
is equivalent to
mknod (path, (mode & 0777) | S_IFIFO, 0)

Return Value
Upon successful completion, a value of 0 is returned. If the **mknod** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **mknod** system call fails and the new file is not created if one or more of the following are true:

EPERM The process's effective user ID is not superuser and the filetype is not **S_IFIFO**.

ENOTDIR A component of the path prefix is not a directory.

ENOENT A component of the path prefix does not exist or the **path** parameter points to an empty string.

EROFS The directory in which the file is to be created is located on a read-only file system.

EEXIST The named file exists.

EFAULT The **path** parameter points to a location outside of the process's allocated address space.

ESTALE The process's root or current directory is located in an NFS virtual file system that has been unmounted.

ENAMETOOLONG
A component of the **path** parameter exceeded **NAME_MAX** characters or the entire **path** parameter exceeded **PATH_MAX** characters.

EACCES The **mode** parameter specified a directory and the **path** parameter explicitly named a hidden directory component.

EACCES Search permission is denied for a component of the path.

EACCES The directory in which the file is to be created does not permit writing.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

ELOOP A loop of symbolic links was detected.

ENOSPC The file system is out of inodes or the directory in which the regular file, special file, FIFO special file, or directory was to have been created does not have room for the new entry and cannot be extended.

EDQUOT The directory in which the entry for the new link is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

If the Transparent Computing Facility is installed on your system, **mknod** can also fail if one or more of the following are true:

ESITEDN1 **path** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE **path** is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of **path** is replicated but not stored on any site which is currently up.

EROFS Write access is requested for a file on a replicated file system in which the primary copy is unavailable.

EPERM A site value out of the range 0 through 31 was specified with a block or character special mode value (**mknodx** only).

EINTR A signal was caught during the system call.

Related Information
In this book: "bind" in topic 1.2.20, "chhidden" in topic 1.2.42, "chmod, fchmod" in topic 1.2.44, "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, "mkdir" in topic 1.2.168, "umask" in topic 1.2.314, "symlink" in topic 1.2.294, "fs" in topic 2.3.20, and "stat.h" in topic 2.4.22.

The **chmod**, **mkdir**, and **mknod** commands in *AIX Operating System Commands Reference*.

1.2.170 mktemp

Purpose
Constructs a unique file name.

Library
Standard C Library (**libc.a**)

Syntax

```
char *mktemp (template)
char *template;
```

Description
The **mktemp** subroutine replaces the contents of the string pointed to by the **template** parameter with a unique file name.

The string in the **template** parameter must be a file name with six trailing **X**s. The **mktemp** subroutine replaces the **X**s with a character sequence such that the string pointed to by **template** is a filename which does not name an existing file.

Upon successful completion, the **mktemp** subroutine returns the address of the string pointed to by the **template** parameter.

If **mktemp** is unable to construct a unique file name from the template, then the first character of the **template** string is replaced with a null character, and a NULL pointer is returned.

Related Information
In this book: "getpid, getpgrp, getppid" in topic 1.2.110, "tmpfile" in topic 1.2.305, and "tmpnam, tempnam" in topic 1.2.306.

1.2.171 monitor, monstartup, moncontrol

Purpose
Starts and stops execution profiling.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <mon.h>

void monitor (lowpc, highpc, shortbuff, bufsize, nfunc)
-- or --
void monitor (lowpc, highpc, profbuff, -1, nfunc)
```

Note: The **lowpc** value for the second definition must be nonzero.

```
int (*lowpc) ( ), (*highpc) ( );
short *shortbuff;
struct prof *profbuff;
int bufsize, nfunc;

int monstartup (lowpc, highpc)
int (*lowpc) ( ), (*highpc) ( );
void moncontrol (mode)
int mode;
```

Description
The **monitor** subroutine records a histogram of periodically sampled values of the program counter and counts the number of times certain subroutines are called. The **monitor** subroutine is an interface to the **profil** system call.

Executable programs created with **cc -p** automatically include calls to the **monitor** subroutine. You do not need to call the **monitor** subroutine unless you want fine control over profiling.

If the **bufsize** parameter has any value other than -1, then the parameters to **monitor** are interpreted as shown in the first syntax definition. The **lowpc** parameter specifies the lowest address to be sampled, and the highest address to be sampled is the address just below **highpc**. The **lowpc** parameter cannot be 0 when using the **monitor** subroutine to begin profiling. If **monitor** is called with a **lowpc** value of 0, then monitoring is stopped and the results are written to a file named **mon.out**.

The **shortbuff** parameter points to a user-supplied array of short integers. The number of **short**s in **shortbuff** is specified by the **bufsize** parameter.

The **nfunc** parameter specifies the maximum number of subroutines whose calls are to be counted. Only calls to functions compiled with the **-p** flag of the **cc** command are recorded.

For the results to be significant, especially for programs with small, heavily-used subroutines, specify a buffer that is no more than a few times smaller than the range of locations sampled.

If **bufsize** has the value -1, then the parameters to **monitor** are interpreted as shown in the second syntax definition. In this case, the

arguments **lowpc** and **highpc** are ignored, **nfunc** retains the same meaning as described above, and **profbuff** points to an array of **prof** structures. The **prof** structure is defined in the **mon.h** header file, and it contains the following members:

```
daddr_t  p_low;
daddr_t  p_high;
short_t  *p_buff;
int_t    p_bufsize;
int_t    p_scale;
```

The **monitor** subroutine ignores the value given in **p_scale** and computes a value for it. The **p_high** members in successive structures must be in ascending sequence. The array of structures is terminated with a structure containing a **p_high** member set to 0.

Use the **prof** command to examine the results after executing your program.

The **monstartup** subroutine is a high level interface to **profil**. The **lowpc** and **highpc** parameters specify the address range that is to be sampled; the lowest address sampled is that of **lowpc** and the highest is just below **highpc**. The **monstartup** subroutine allocates space using **sbrk** and passes it to **monitor** to record a histogram of periodically sampled values of the program counter, and of counts of calls to certain functions, in the buffer. Only calls of functions compiled with the profiling option **-p** of **cc** are recorded.

To profile the entire program, it is sufficient to use the following:

```
extern etext();
...
monstartup((int)2, etext);
```

The **etext** parameter points to just above all the program text.

To stop execution monitoring and write the results on the file **mon.out**, use the following:

```
monitor(0);
```

Then use **prof** to examine the results.

The **moncontrol** subroutine is used to selectively control profiling within a program. This works with either **prof** or **gprof** profiling. When the program starts, profiling begins. This allows the cost of a particular operation to be measured. Note that an output file is produced upon program exit, regardless of the state of **moncontrol**.

Examples

1. To profile the entire AIX PS/2 program:

```
extern etext;
...
monitor ((int (*)()) 0x00400000, etext, buf, bufsize, nfunc);
```

   The identifier **etext** is the address immediately following the program text. (See "end, etext, edata" in topic 1.2.68 for more information about **etext**.)

2. To profile the entire AIX System/370 program:

```
extern etext;
...
monitor ((int (*)()) 0x00010000, etext, buf, bufsize, nfunc);
```

   The identifier **etext** is the address immediately following the program text. (See "end, etext, edata" in topic 1.2.68 for more information about **etext**.)

3. To profile an entire AIX PS/2 program that includes a shared library:

```
extern etext;
struct prof buf[3];
...

buf[0].p_low  = 0x00400000    /* program text        */
buf[0].p_high = etext

buf[1].p_low  = 0xD0000000    /* shared library text */
buf[1].p_high = 0xD0030D40

buf[2].p_low  = 0             /* end of array        */
buf[2].p_high = 0

monitor((int (*)())0, (int (*)())0, buf, -1, nfunc);
```

   The addresses shown for the shared library text may differ from the ones appropriate for a program you write.

4. To stop execution monitoring and write the results to the file **mon.out**:

```
monitor ((int (*)()) 0, (int (*)())0, (short *)0, 0, 0);
```

File
mon.out

Related Information
In this book: "end, etext, edata" in topic 1.2.68 and "profil" in topic 1.2.210.

The **cc** and **prof** commands in *AIX Operating System Commands Reference*.

1.2.172 mount

Purpose
Mounts a file system.

Syntax

```
#include <sys/vmount.h>

int mount (dev, dir, mflag)
char *dev, *dir;
int mflag;
```

Description
The **mount** system call mounts a file system contained on the block device (also called a ***special file***) identified by the **dev** parameter. The file system is mounted on the directory identified by the **dir** parameter. The **mount** system call can be used only by the superuser.

The **dev** parameter and the **dir** parameter are pointers to path names.

The **mflag** parameter defines various characteristics of the object to be mounted. A possible value is:

MNT_READONLY Indicates that the object to be mounted is read-only, and write access is not allowed. If this value is not specified, writing is permitted according to individual file accessibility.

After the file system is mounted, references to the path name specified by the **dir** parameter refer to the root directory on the mounted file system.

If the Transparent Computing Facility is installed, separate copies of a replicated file system must be mounted on the same directory and with the same value of **mflag**. If the primary copy of the file system is not mounted, the file system is treated as read-only.

Return Value
Upon successful completion a value of 0 is returned. If the **mount** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **mount** system call fails if one or more of the following are true:

EPERM The effective user ID of the calling process is not superuser.

ENOENT **dev** or **dir** does not exist.

ENOTBLK **dev** is not a block device.

ENXIO The device or driver for **dev** is not currently configured.

ENOTDIR A component of a path prefix is not a directory.

ENOTDIR **dir** is not a directory.

EFAULT The **dev** or **dir** parameter points to a location outside of the process's allocated address space.

EBUSY **dir** is currently busy. For example, a file system may be mounted onto it.

EBUSY The device associated with **dev** is currently mounted.

EBUSY The file system number corresponding to **dev** is in use by another mounted device.

EINVAL The data on **dev** is not recognizable as a file system. This usually means that it does not contain a properly formatted super block.

ENAMETOOLONG
A component of the **path** parameter exceeded **NAME_MAX** characters or the entire **path** parameter exceeded **PATH_MAX** characters.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

ELOOP A loop of symbolic links was detected.

EACCES Search permission is denied for a component of **dev** or **dir**.

EIO An I/O error occurred in performing the mount.

If the Transparent Computing Facility is installed on your system, **mount** can also fail if one or more of the following are true:

ENOSTORE **dev** or **dir** is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of **dir** is replicated but not stored on any site which is currently up.

EROFS An attempt was made to mount a copy of a replicated file system with an **mflag** value which differed from that of the other currently mounted copies.

ENLDEV **dev** is a remote device.

EXGFS An attempt was made to mount a file system whose file system number was already in use in an inconsistent way. The problem is that neither copy was replicated, the type of replication was not the same for both file systems, or they were not mounted at the same mount point.

EPBUSY An attempt was made to mount a copy of a replicated file system which had a pack number that was already mounted.

Related Information
In this book: "umount, fumount" in topic 1.2.315 and "fs" in topic 2.3.20.

The **mount** and **umount** commands in *AIX Operating System Commands Reference*.

1.2.173 msgctl

Purpose
Provides message control operations.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid.ds *buf;
```

Description
The **msgctl** system call provides a variety of message control operations as specified by **cmd** parameter. The **buf** parameter points to a structure of type **msqid.ds**. The **msqid.ds** structure is defined in the **sys/msg.h** header file, and it contains the following members:

```
  struct ipc_perm  msg_perm;    /* Operation permission structure */
  struct msg      *msg_first;   /* Ptr to first message on the queue */
  struct msg      *msg_last;    /* Ptr to last message on the queue */
  ushort           msg_cbytes;  /* Current number of bytes on the queue */
  ushort           msg_qnum;    /* Number of messages on the queue */
  ushort           msg_qbytes;  /* Maximum number of bytes on the queue */
  pid_t           msg_lspid;    /* ID of last process to call msgsnd */
  pid_t           msg_lrpid;    /* ID of last process to call msgrcv */
  time_t          msg_stime;    /* Time of last msgsnd call */
  time_t          msg_rtime;    /* Time of last msgrcv call */
  time_t          msg_ctime;    /* Time of the last change to this */
                                /*  structure with a msgctl call */
```

The following **cmd**s are available:

IPC_STAT Stores the current value of the members of the data structure associated with the **msqid** parameter into the **msqid.ds** structure pointed to by the **buf** parameter. The current process must have read permission in order to perform this operation.

IPC_SET Sets the value of the following members of the data structure associated with the **msqid** parameter to the corresponding values found in the structure pointed to by the **buf** parameter:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode  /* Only the low-order nine bits */
msg_qbytes
```

The current process must have an effective user ID equal to either that of superuser or to the value of **msg_perm.uid** in the data structure associated with **msqid** in order to perform this operation. To raise the value of **msg_qbytes**, the effective user ID of the current process must be superuser.

IPC_RMID Removes the message queue identifier specified by the **msqid**

parameter from the system and destroys the message queue and data structure associated with it. The current process must have an effective user ID equal to either that of superuser or to the value of **msg_perm.uid** in the data structure associated with **msqid** in order to perform this operation.

Warning: If the Transparent Computing Facility is installed, a message queue exists only on the cluster site on which the queue was created. Consequently, processes that use message queues cannot be migrated to other sites.

Return Value
Upon successful completion, a value of 0 is returned. If the **msgctl** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **msgctl** system call fails if one or more of the following are true:

EINVAL **msqid** is not a valid message queue identifier.

EINVAL **cmd** is not a valid command.

EACCES **cmd** is equal to **IPC_STAT** and read permission is denied to the calling process.

EPERM **cmd** is equal to **IPC_RMID**, **IPC_SET**, or **IPC_RMID2** and the effective user ID of the calling process is not equal to that of superuser, nor is it equal to the value of **msg_perm.uid** in the data structure associated with **msqid**.

EPERM **cmd** is equal to **IPC_SET**, an attempt is being made to increase to the value of **msg_qbytes**, and the effective user ID of the calling process is not equal to that of superuser.

EFAULT The **buf** parameter points to a location outside of the process's allocated address space.

Related Information
In this book: "msgget" in topic 1.2.174, "msgrcv" in topic 1.2.178, "msgsnd" in topic 1.2.180, and "msgxrcv" in topic 1.2.181.

1.2.174 msgget

Purpose
Gets a message queue identifier.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key, msgflg)
key_t key;
int msgflg;
```

Description
The **msgget** system call returns the message queue identifier associated with the specified **key**. The **key** parameter is either the value **IPC_PRIVATE** or an IPC key constructed by the **ftok** subroutine (or by a similar algorithm). See " stdipc: ftok" in topic 1.2.284 for details about this subroutine.

The **msgflg** parameter is constructed by logically ORing one or more of the following values:

IPC_CREAT	Creates the data structure if it does not already exist.
IPC_EXCL	Causes the **msgget** system call to fail if **IPC_CREAT** is also set and the data structure already exists.
S_IRUSR	Permits the process that owns the data structure to read it.
S_IWUSR	Permits the process that owns the data structure to modify it.
S_IRGRP	Permits the group associated with the data structure to read it.
S_IWGRP	Permits the group associated with the data structure to modify it.
S_IROTH	Permits others to read the data structure.
S_IWOTH	Permits others to modify the data structure.

The values that begin with **S_I**- are defined in the **sys/stat.h** header file and are a subset of the access permissions that apply to files.

A message queue identifier and associated message queue and data structure are created for the value of the **key** parameter if one of the following are true:

key is equal to **IPC_PRIVATE**.

key does not already have a message queue identifier associated with it, and **IPC_CREAT** is set.

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

msg_perm.cuid, **msg_perm.uid**, **msg_perm.cgid**, and **msg_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order nine bits of **msg_perm.mode** are set equal to the low-order nine bits of the **msgflg** parameter.

msg_qnum, **msg_lspid**, **msg_lrpid**, **msg_stime**, and **msg_rtime** are set equal to 0.

msg_ctime is set equal to the current time.

msg_qbytes is set equal to the system limit.

Warning: If the Transparent Computing Facility is installed, a message queue exists only on the cluster site on which the queue was created. There is no provision for accessing a message queue on a remote cluster site. Consequently, processes that use message queues cannot be migrated to other sites.

Return Value
Upon successful completion, a message queue identifier is returned. If the **msgget** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **msgget** system call fails if one or more of the following are true:

EACCES A message queue identifier exists for the **key** parameter but operation permission as specified by the low-order nine bits of the **msgflg** parameter would not be granted.

ENOENT A message queue identifier does not exist for the **key** parameter and **IPC_CREAT** is not set.

ENOSPC A message queue identifier is to be created but the system imposed limit on the maximum number of allowed message queue identifiers system wide would be exceeded.

EEXIST A message queue identifier exists for **key**, and both **IPC_CREAT** and **IPC_EXCL** are set.

Related Information
In this book: "msgctl" in topic 1.2.173, "msgrcv" in topic 1.2.178, "msgsnd" in topic 1.2.180, "msgxrcv" in topic 1.2.181, and " stdipc: ftok" in topic 1.2.284.

1.2.175 msghelp

Purpose
Issues help text.

Library
Run-time Services Library (**librts.a**)

Syntax

```
#include <msg00.h>

int msghelp (flags, compid, index [, fildes])
unsigned int flags;
char *compid;
int index, fildes;
```

Description
The **msghelp** subroutine retrieves a predefined help description from a **message/insert/help** file and then constructs and outputs the help text.

The **flags** parameter allows default help attributes to be overridden. All flag bits for attributes you do not want to override must be off. If no attributes are overridden, the help is written to **stderr**. Attribute override flag bits that can be set are:

MSGFLFIL Writes the help text to the file specified by the **fildes** parameter. If this flag is not set, then the help text is written to **stderr**.

There is no specific flag bit defined for suppressing output of the help ID. If you want to suppress the help ID, do not specify the displayed component ID and displayed help ID fields of the help description in the **message/insert/help** file. If the help ID is suppressed, then the help text is aligned **fildes** parameter causes the help text to be aligned at the left margin instead of to the right of the help ID. This allows a full 79-character width, but does not provide component and help IDs for referencing an explanation of the help in a reference manual.

The **compid** parameter points to a six-character string that identifies the **message/insert/help** file where the help control information resides. The **compid** parameter is either:

xxxccc For a component file, where, by convention:

- *xxx* Identifies the software provider or product. IBM reserves the use of the identifiers **COM**, **com**, **SYc**, **syc**, **IBc**, and **ibc**, where **c** is any alphanumeric character.
- *ccc* Identifies the particular software component.

common For the common **message/insert/help** file.

The **index** parameter is an index into the file specified by the **compid** parameter. The **index** parameter is an integer value from 1 to 999 and identifies which help description in the file is to be used.

The **fildes** parameter is an integer file descriptor number indicating the opened file to which the help is to be sent. The **fildes** parameter is used only if the **MSGFLFIL** flag is on.

Return Value
Upon successful completion, a value of 0 is returned. If the **msghelp** subroutine fails, then it returns one of the following negative values.

The following values are defined in the **msg04.h** header file, which is included by the **msg00.h** header file:

MSG_CPID The **compid** parameter is not six characters long. The request is ignored.

MSG_INDX The **index** parameter is not in the range of 1 to 999. The request is ignored.

MSG_TABP The **MSGFLTAB** flag is on. Since helps cannot reside in a message/insert table, this is not a valid flag for the **msghelp** subroutine. The request is ignored.

MSG_ALLO The necessary Message Services work area cannot be allocated. The request is ignored.

MSG_SREG A segment register is not available for mapping a **message/insert/help** file. The request is ignored.

MSG_COMP The **message/insert/help** file specified by the **compid** parameter cannot be found. Message Services error message 090-002 is output instead.

MSG_INVL The file specified by the **compid** parameter is not a valid **message/insert/help** file. Message Services error message 090-002 is output instead.

MSG_MTCH The file specified by the **compid** parameter does not contain descriptions for the specified component. The first six characters of the component file name must be identical to the six-character component ID that was specified in the file to the **puttext** command when the component file was built. Message Services error message 090-002 is output instead.

MSG_NONE The correct component files are found, but none contain the message description specified by the **index** parameter. Message Services error message 090-002 is output instead.

MSG_REFN The requested help description is found but the description references another help description (in the same file) as the source of the text. The referenced help description does not exist. Message Services error message 090-002 is output instead.

Note: Certain errors involve the failure of AIX system calls. In these cases, the **msghelp** subroutine negates the error code that the system call stored in **errno** and returns this value.

One of the following values is returned when an attempt to open a **message/insert/help** file fails:

-EACCES Search permission is denied for a directory in the path prefix of the **message/insert/help** file.

-ENOTDIR A component of the path name of the **message/insert/help** file

is not a directory.

-EMFILE Too many files are open for the process.

One of the following values is returned when an attempt to write to the file specified by the **fildes** parameter fails:

-EBADF The **fildes** parameter does not specify a valid file descriptor that is open for writing.

-EFBIG The file specified by the **fildes** parameter exceeds the maximum file size or file size limit for the process.

Related Information
In this book: "msgimed" in topic 1.2.176, "msgqued" in topic 1.2.177, "msgrtrv" in topic 1.2.179, and "message" in topic 2.3.33.

1.2.176 msgimed

Purpose
Issues an immediate message.

Library
Run-time Services Library (**librts.a**)

Syntax

#include <msg00.h>

int msgimed (flags, compid, index [**, sevcode** [**, errcode** [**, fildes**]]]**)**
unsigned int flags;
char *compid;
int index, sevcode, fildes;
long errcode;

Description
The **msgimed** subroutine retrieves a predefined message description from a message/insert table or a **message/insert/help** file and then constructs the message text and outputs it.

The **flags** parameter allows default message attributes to be overridden. All flag bits for attributes you do not want to override must be off. If no attributes are overridden, a message consisting of a message ID (if defined) and message text is written to **stderr**. Attribute override flag bits that can be set are:

MSGFLTAB Indicates that the **compid** parameter is a pointer to a message/insert table instead of a pointer to a six-character component ID identifying a **message/insert/help** file.

MSGFLTIM Includes with the message the time the message was issued. The time is given in 24-hour format. This flag should always be set if the error is logged.

MSGFLSEV Includes a severity code with the message. The severity code value is specified by the **sevcode** parameter.

MSGFLERR Includes an error code with the message. The value of the error code is specified by the **errcode** parameter.

MSGFLFIL Writes the message to the file specified by the **fildes** parameter. If this flag is not set, then the message is written to **stderr**.

There is no specific flag bit defined for suppressing output of the message ID. If you want to suppress the message ID, do not specify the **displayed component ID** and the **displayed message ID** fields of the message description in the message/insert table or the **message/insert/help** file. Suppression of the message ID for a message output to **stderr** or to the output specified by the **fildes** parameter causes the message to be aligned at the left margin instead of to the right of the message ID. This allows a full 79-character width, but does not provide component and message IDs for referencing an explanation of the message in a reference manual.

The **compid** parameter is either a pointer to a message/insert table or identifies the **message/insert/help** file. If the **MSGFLTAB** flag is set, then the **compid** parameter is a pointer to a message/insert table where the

message description resides. If the **MSGFLTAB** flag is not set, then the **compid** parameter identifies the **message/insert/help** file where the message description resides. In this case, the **compid** parameter is either:

xxxccc For a component file, where, by convention:

xxx Identifies the software provider or product. IBM reserves the use of the identifiers **COM**, **com**, **SYc**, **syc**, **IBc**, and **ibc**, where **c** is any alphanumeric character.
ccc Identifies the particular software component.

common For the common **message/insert/help** file.

The **index** parameter is an index into the message/insert table or the **message/insert/help** file specified by the **compid** parameter. The **index** parameter is an integer value from 1 to 999 and identifies which message description is to be used.

The **sevcode** parameter specifies an integer severity code that is output with the message if the **msgflerr** flag is set. The following severity codes have been defined:

MSGSVSYT System termination
MSGSVAPT Application termination
MSGSVOPR Operator-recoverable error
MSGSVAPR Application-recoverable error.

If the **MSGFLSEV** flag is not set, and if the **errcode** or **fildes** parameters are specified, then a dummy **sevcode** parameter must be used as a place holder.

The **errcode** parameter is a long integer value that represents an error code with six decimal digits. The error code is output with the message only if the **MSGFLERR** flag is set. The two high-order decimal digits contain the origin code; the four low-order digits contain an application-defined error return code. The origin code is one of the following values:

MSGORIND Indeterminate origin.
MSGORVDD Reserved.
MSGORVCK Reserved.
MSGORVSV Reserved.
MSGORUDD Detected in AIX device driver.
MSGORUKN Detected in AIX kernel.
MSGORSHL Detected in shell command.
MSGORRTS Detected in run-time service or daemon.
MSGORAPP Detected in application above the application program interface.

If the **MSGFLERR** flag is not set, and if the **fildes** parameter is specified, then a dummy **errcode** parameter must be used as a place holder.

The **fildes** parameter is a file descriptor indicating the opened file to which the message is to be sent. The **fildes** parameter is used only if the **msgflfil** flag is set.

Return Value
Upon successful completion, a value of 0 is returned. If the **msgimed** subroutine fails, then it returns one of the following negative values.

The following values are defined in the **msg04.h** header file, which is included by the **msg00.h** header file.

MSG_CPID The **compid** parameter is not six characters long. The request is ignored.

MSG_INDX The **index** parameter is not in the range of 1 to 999. The request is ignored.

MSG_ALLO The necessary Message Services work area cannot be allocated. The request is ignored.

MSG_SREG A segment register is not available for mapping a **message/insert/help** file. The request is ignored.

MSG_BADP The message/insert table pointer provided does not point to a message/insert table. The request is ignored.

MSG_TABI The message/insert table that is provided does not contain the requested message. The request is ignored.

MSG_COMP The **message/insert/help** file specified by the **compid** parameter cannot be found. Message Services error message 090-001 is output instead.

MSG_INVL The file specified by the **compid** parameter is not a valid **message/insert/help** file. Message Services error message 090-001 is output instead.

MSG_MTCH The file specified by the **compid** parameter does not contain descriptions for the specified component. The first six characters of the component file name must be identical to the six-character component ID that is specified in the file to the **puttext** command when the component file was built. Message Services error message 090-001 is output instead.

MSG_NONE The correct component files are found, but none contain the message description specified by the **index** parameter. Message Services error message 090-001 is output instead.

MSG_REFN The requested message description is found but the description references another message description (in the same file) as the source of the text. The referenced message description does not exist. Message Services error message 090-001 is output instead.

Note: Certain errors involve the failure of AIX system calls. In these cases, the **msghelp** subroutine negates the error code that the system call stored in **errno** and returns this value.

One of the following values is returned when an attempt to open a **message/insert/help** file fails:

-EACCES Search permission is denied for a directory in the path prefix. The request is ignored.

-ENOTDIR A component of the path prefix is not a directory. The request is ignored.

-EMFILE Too many files are open for the process. The request is ignored.

One of the following values is returned when an attempt to write to the file specified by the **fildes** parameter fails:

-EBADF Not a valid file descriptor open for writing.

-EFBIG The file exceeds the process's file size limit or the maximum file size.

Related Information
In this book: "msghelp" in topic 1.2.175, "msgqued" in topic 1.2.177, "msgrtrv" in topic 1.2.179, and "message" in topic 2.3.33.

1.2.177 msgqued

Purpose
Issues a queued message.

Library
Run-time Services Library (**librts.a**)

Syntax

```
#include <msg00.h>

int msgqued (flags, compid, index [, sevcode [, errcode]])
unsigned int flags;
char *compid;
int index, sevcode;
long errcode;
```

Description
The **msgqued** subroutine retrieves a predefined message description from a message/insert table or a **message/insert/help** file and then constructs the message text and writes it to the queued message file, **/qmsg**.

The queued message file is installed with the AIX Operating System. After installation, you can change the default size of the queued message file (six 2048-byte blocks) by using an editor to modify the four-digit value between the first two asterisks (*****) in the first line of the file. This four-digit value is in units of 2048-byte blocks.

After **/qmsg** reaches the size specified in the first line, each new message added to the queue overlays the oldest message in the file. The message queue is maintained across IPLs.

Queued messages are directed to the console operator and are generally system type messages.

The **flags** parameter allows default message attributes to be overridden. All flag bits for attributes you do not want to override must not be set. If no attributes are overridden, then the message consists of the message ID (if defined), the message text, and the date and time the message was issued. Attribute override flag bits that can be set are:

MSGFLTAB Indicates that the **compid** parameter is a pointer to a message/insert table instead of a pointer to a six-character component ID identifying a **message/insert/help** file.

MSGFLSEV Includes a severity code with the message. The severity code value is specified by the **sevcode** parameter.

MSGFLERR Includes an error code with the message. The error code value must be specified by the **errcode** parameter.

The **compid** parameter is either a pointer to a message/insert table or identifies the **message/insert/help** file. If the **MSGFLTAB** flag is set, then the **compid** parameter points to a message/insert table where the message description resides. If the **MSGFLTAB** flag is not set, the **compid** parameter identifies the **message/insert/help** file where the message description resides.

The **index** parameter is an index into the message/insert table or

message/insert/help file specified by the **compid** parameter. The **index** parameter is an integer value from 1 to 999 and identifies which message description in the file is to be used.

The **sevcode** parameter specifies an integer severity code that is written with the message if the **MSGFLERR** flag is set. The following severity codes have been defined:

MSGSVSYT System termination
MSGSVAPT Application termination
MSGSVOPR Operator-recoverable error
MSGSVAPR Application-recoverable error.

If the **msgflsev** flag is not set, and if the **errcode** parameter is specified, then a dummy **sevcode** parameter must be used as a place holder.

The **errcode** parameter is a long integer value that represents an error code with six decimal digits. The error code is output with the message only if the **MSGFLERR** flag is set. The two high-order decimal digits contain the origin code; the four low-order digits contain an application-defined error return code. The possible values for the origin code are listed in the description. The origin code is one of the following values:

MSGORIND Indeterminate origin.
MSGORVDD Reserved.
MSGORVCK Reserved.
MSGORVSV Reserved.
MSGORUDD Detected in AIX device driver.
MSGORUKN Detected in AIX kernel.
MSGORSHL Detected in shell command.
MSGORRTS Detected in run-time service or daemon.
MSGORAPP Detected in application above the application program interface.

Return Value
Upon successful completion, a value of 0 is returned. If the **msgqued** subroutine fails, then it returns one of the following negative values.

The following values are defined in the **msg04.h** header file, which is included by the **msg00.h** header file:

MSG_CPID The **compid** parameter is not six characters long. The **msgqued** request is ignored.

MSG_INDX The **index** parameter is not in the range of 1 to 999. The **msgqued** request is ignored.

MSG_ALLO The necessary Message Services work area cannot be allocated. The **msgqued** request is ignored.

MSG_SREG A segment register is not available for mapping a **message/insert/help** file. The **msgqued** request is ignored.

MSG_BADP The message/insert table pointer provided does not point to a message/insert table. The **msgqued** request is ignored.

MSG_TABI The message/insert table that is provided does not contain the requested message. The **msgqued** request is ignored.

MSG_COMP The **message/insert/help** file specified by the **compid** parameter cannot be found. Message Services error message 090-001 is output instead.

MSG_INVL The file specified by the **compid** parameter is not a valid **message/insert/help** file. Message Services error message 090-001 is output instead.

MSG_MTCH The file specified by the **compid** parameter does not contain descriptions for the specified component. The first six characters of the component file name must be identical to the six-character component ID that is specified in the file to the **puttext** command when the component file was built. Message Services error message 090-001 is output instead.

MSG_NONE The correct component files are found, but none contain the message description specified by the **index** parameter. Message Services error message 090-001 is output instead.

MSG_REFN The requested message description is found but the description references another message description (in the same file) as the source of the text. The referenced message description does not exist. Message Services error message 090-001 is output instead.

MSG_EXEC The **fork** or **exec** system call failed while attempting to run the program that updates the queued message file. The **msgqued** request is ignored. The failure of **exec** is not detected if the calling process catches the **SIGCLD** signal. See "sigaction, sigvec, signal" in topic 1.2.263 about catching signals and the special handling of **SIGCLD**.

MSG_QMSG The queued message file, **/qmsg**, cannot be opened, or its format is not valid. The **msgqued** request is ignored. This condition is not detected if the calling process catches the **SIGCLD** signal. See "sigaction, sigvec, signal" in topic 1.2.263 about catching signals and the special handling of **SIGCLD**.

Note: Certain errors involve the failure of AIX system calls. In these cases, the **msghelp** subroutine negates the error code that the system call stored in **errno** and returns this value.

One of the following values is returned when an attempt to open a **message/insert/help** file fails:

-EACCES Search permission is denied for a directory in the path prefix of the **message/insert/help** file.

-ENOTDIR A component of the path name of the **message/insert/help** file is not a directory.

-EMFILE Too many files are open for the process.

File

/qmsg

Related Information
In this book: "msghelp" in topic 1.2.175, "msgimed" in topic 1.2.176,

"msgrtrv" in topic 1.2.179, and "message" in topic 2.3.33.

1.2.178 msgrcv

Purpose
Reads a message from a queue.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

Description
The **msgrcv** system call reads a message from the queue specified by the **msqid** parameter and stores it into the structure pointed to by the **msgp** parameter. The current process must have read permission in order to perform this operation. The **msgbuf** structure is defined in the **sys/msg.h** header file, and it contains the following members:

```
long  mtype;     /* Message type */
char  mtext[1];  /* Beginning of message text */
```

The **mtype** field contains the type of the received message as specified by the sending process. **mtext** is the text of the message.

The **msgsz** parameter specifies the size of **mtext** in bytes. The received message is truncated to the size specified by the **msgsz** parameter if it is longer than the size specified by the **msgsz** parameter and if **MSG_NOERROR** is set in **msgflg**. The truncated part of the message is lost and no indication of the truncation is given to the calling process. If the message is longer than **msgsz** bytes and **MSG_NOERROR** is not set, then the **msgrcv** system call fails and sets **errno** to E2BIG.

The **msgtyp** parameter specifies the type of message requested as follows:

If the **msgtyp** parameter is equal to 0, the first message on the queue is received.

If the **msgtyp** parameter is greater than 0, the first message of the type specified by the **msgtyp** parameter is received.

If the **msgtyp** parameter is less than 0, the first message of the lowest type that is less than or equal to the absolute value of the **msgtyp** parameter is received.

The **msgflg** parameter is either 0, or is constructed by logically ORing one or more of the following values:

MSG_NOERROR Truncates the message if it is longer than **msgsz** bytes.

IPC_NOWAIT Specifies the action to take if a message of the desired type is not on the queue:

If **IPC_NOWAIT** is set, then the calling process returns a value of -1 and sets **errno** to ENOMSG.

If **IPC_NOWAIT** is not set, then the calling process suspends execution until one of the following occurs:

- A message of the desired type is placed on the queue.
- The message queue identifier specified by the **msqid** parameter is removed from the system. When this occurs, **errno** is set to EIDRM, and a value of -1 is returned.
- The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes in the manner described in "sigaction, sigvec, signal" in topic 1.2.263.

Warning: If the Transparent Computing Facility is installed, a message queue exists only on the cluster site on which the queue was created. There is no provision for accessing a message queue on a remote cluster site. Consequently, processes that use message queues cannot be migrated to other sites.

Return Value

Upon successful completion, **msgrcv** returns a value equal to the number of bytes actually stored into **mtext** and the following actions are taken with respect to the data structure associated with the **msqid** parameter:

msg_qnum is decremented by 1.
msg_lrpid is set equal to the process ID of the calling process.
msg_rtime is set equal to the current time.

If the **msgrcv** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions

The **msgrcv** system call fails if one or more of the following are true:

EINVAL **msqid** is not a valid message queue identifier.

EACCES Operation permission is denied to the calling process.

EINVAL **msgsz** is less than 0.

E2BIG **mtext** is greater than **msgsz** and **MSG_NOERROR** is not set.

ENOMSG The queue does not contain a message of the desired type and **IPC_NOWAIT** is set.

EFAULT The **msgp** parameter points to a location outside of the process's allocated address space.

EINTR **msgrcv** received a signal.

EIDRM The message queue identifier specified by **msqid** has been removed from the system.

Related Information
In this book: "msgctl" in topic 1.2.173, "msgget" in topic 1.2.174, "msgsnd" in topic 1.2.180, "msgxrcv" in topic 1.2.181, and "sigaction, sigvec, signal" in topic 1.2.263.

1.2.179 msgrtrv

Purpose
Retrieves a message, insert, or help text.

Library
Run-time Services Library (**librts.a**)

Syntax

```
#include <msg00.h>

int msgrtrv (flags, compid, index, buf, nbytes)
unsigned int flags, nbyte;
char *compid, *buf;
int index;
```

Description
The **msgrtrv** subroutine retrieves a predefined message, insert, or help description from a **message/insert/help** file or a resident message/insert table, and then constructs the message, insert, or help text as specified and returns the text.

The **flags** parameter allows default attributes to be overridden. All flag bits for attributes you do not want to override must not be set. If no attributes are overridden, insert text is retrieved from a file. Attribute override flag bits that can be set are:

MSGFLTAB Indicates that the **compid** parameter is a pointer to a message/insert table instead of a pointer to a six-character component ID identifying a **message/insert/help** file. The **MSGFLTAB** flag should not be set if the **MSGFLHLP** flag is set because helps reside only in a **message/insert/help** file, not in a message/insert table.

MSGFLMSG Retrieves message text instead of insert text.

MSGFLHLP Retrieves help text instead of insert text.

The **compid** parameter is either a pointer to a message/insert table or identifies the **message/insert/help** file. If the **MSGFLTAB** flag is set, then the **compid** parameter points to a message/insert table where the message or insert description resides. If the **MSGFLTAB** flag is not set, then the **compid** parameter identifies the **message/insert/help** file where the message, insert, or help description resides. In this case, the **compid** parameter is either:

xxxccc For a component file, where, by convention:

- *xxx* Identifies the software provider or product. IBM reserves the use of the identifiers **COM**, **com**, **SYc**, **syc**, **IBc**, and **ibc**, where **c** is any alphanumeric character.
- *ccc* Identifies the particular software component.

common For the common **message/insert/help** file.

The **index** parameter is an index into the message/insert table or **message/insert/help** file specified by the **compid** parameter. The **index** parameter is an integer value from 1 to 999 and identifies which message, insert, or help description in the file or table is to be used.

The **buf** parameter must be either a pointer to a buffer or a pointer to a structure, depending on the value of the **nbyte** parameter.

If the **nbyte** parameter is greater than 0, then **buf** parameter points to a buffer where the message, insert, or help text is to be stored.

If the **nbyte** parameter is equal to 0, then the **buf** parameter points to a **msg__rtrv** structure provided by the requesting program. The **msg__rtrv** is defined as a **typedef** in the **msg05.h** header file.

The **nbyte** parameter is either the size of the buffer pointed to by the **buf** parameter, or 0. The buffer size should include space for a terminating null character. If the **nbyte** parameter is 0, a buffer is allocated by the **msgrtrv** subroutine. The buffer pointer (**msgbufp** in the **msg05.h** header file) returned by the **msgrtrv** subroutine should always be inspected by the requesting program after the returned text has been processed. If the inspection finds other than a NULL buffer pointer, the buffer should be freed. This should be done regardless of the value of the return code.

Return Value
Upon successful completion, a positive value is returned. If the **msgrtrv** subroutine fails, it returns a negative value that indicates the reason why the text could not be retrieved.

The value returned upon successful completion is the actual length of the constructed text, not including the terminating null character. The following should be noted concerning the length:

If the **nbyte** parameter was 0 and help text with a title was retrieved, the length returned is the sum of the title length and the text length, including the null terminators after the title and the text.

If the **nbyte** parameter was not 0, and the retrieved text is longer than the buffer provided (minus 1 character for the null terminator), the excess text is truncated. The length of the truncated text is included in the length returned. If the return code value is greater than the length specified by the **nbyte** parameter minus 1, the following considerations should be noted:

- The length of the text returned in the buffer is the length specified by the **nbyte** parameter minus one instead of the return code value.

- The requesting program knows that the retrieved text had to be truncated in order to fit into the buffer provided.

If the **msghelp** subroutine fails, then it returns one of the following negative values.

The following values are defined in the **msg04.h** header file, which is included by the **msg00.h** header file:

MSG_CPID The **compid** parameter is not six characters long. The request is ignored.

MSG_INDX The **index** parameter is not in the range of 1 to 999. The request is ignored.

MSG_TABP Both the **msgfltab** and **msgflhlp** flags are on. Since helps

cannot reside in a message/insert table, this is not a valid combination of flag bits. The request is ignored.

MSG_ALLO The necessary Message Services work area cannot be allocated. The request is ignored.

MSG_SREG A segment register is not available for mapping a **message/insert/help** file. The request is ignored.

MSG_BADP The message/insert table pointer provided does not point to a message/insert table. The request is ignored.

MSG_TABI The message/insert table that is provided does not contain the requested message or insert. The request is ignored.

MSG_COMP The **message/insert/help** file specified by the **compid** parameter cannot be found. If a message was specified, then Message Services error message 090-001 is output instead. If an insert was specified, then the request is ignored. If help text was specified, then Message Services error message 090-002 is output instead.

MSG_INVL The file specified by the **compid** parameter is not a valid **message/insert/help** file. If a message was specified, then Message Services error message 090-001 is output instead. If an insert was specified, then the request is ignored. If help text was specified, then Message Services error message 090-002 is output instead.

MSG_MTCH The file specified by the **compid** parameter does not contain descriptions for the specified component. The first six characters of the component file name must be identical to the six-character component ID that was specified in the file to the **puttext** command when the component file was built. If a message was specified, then Message Services error message 090-001 is output instead. If an insert was specified, then the request is ignored. If help text was specified, then Message Services error message 090-002 is output instead.

MSG_NONE The correct component files are found, but none contain the message, insert, or help description specified by the **index** parameter. If a message was specified, then Message Services error message 090-001 is output instead. If an insert was specified, then the request is ignored. If help text was specified, then Message Services error message 090-002 is output instead.

MSG_REFN The requested message, insert, or help description is found but the description references another message, insert, or help description (in the same file) as the source of the text. The referenced message, insert, or help description does not exist. If a message was specified, then Message Services error message 090-001 is output instead. If an insert was specified, the request is ignored. If help text was specified, then Message Services error message 090-002 is output instead.

Note: Certain errors involve the failure of AIX system calls. In these cases, the **msghelp** subroutine negates the error code that the system call stored in **errno** and returns this value.

One of the following values is returned when an attempt to open a **message/insert/help** file fails:

-EACCES Search permission is denied for a directory in the path prefix of the **message/insert/help** file.

-ENOTDIR A component of the path name of the **message/insert/help** file is not a directory.

-EMFILE Too many files are open for the process.

Related Information
In this book: "msghelp" in topic 1.2.175, "msgimed" in topic 1.2.176, "msgqued" in topic 1.2.177, and "message" in topic 2.3.33.

1.2.180 msgsnd

Purpose
Sends a message.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz, msgflg;
```

Description
The **msgsnd** system call sends a message to the queue specified by the **msqid** parameter. The current process must have write permission in order to perform this operation. The **msgp** parameter points to a **msgbuf** structure containing the message. The **msgbuf** structure is defined in the **sys/msg.h** header file, and it contains the following members:

```
long  mtype;     /* Message type */
char  mtext[1];  /* Beginning of message text */
```

The **mtype** parameter is a positive integer that is used by the receiving process for message selection. The **mtext** parameter is any text of the length in bytes specified by the **msgsz** parameter. The **msgsz** parameter can range from 0 to a system-imposed maximum.

The **msgflg** parameter specifies the action to be taken if the message cannot be sent for one of the following reasons:

The number of bytes already on the queue is equal to **msg_qbytes**.

The total number of messages on all queues system-wide is equal to system-imposed limit.

These actions are as follows:

If **msgflg** is set to **IPC_NOWAIT**, then the message is not sent, and **msgsnd** returns a value of -1 and sets **errno** to EAGAIN.

If **msgflg** is 0, then the calling process suspends execution until one of the following occurs:

- The condition responsible for the suspension no longer exists, in which case the message is sent.
- **msqid** is removed from the system. (For information on how to remove **msqid**, see "msgctl" in topic 1.2.173.) When this occurs, **errno** is set equal to EIDRM, and a value of -1 is returned.
- The calling process receives a signal that is to be caught. In this case the message is not sent and the calling process resumes

execution in the manner prescribed in "sigaction, sigvec, signal" in topic 1.2.263.

Warning: If the Transparent Computing Facility is installed, a message queue exists only on the cluster site on which the queue was created. There is no provision for accessing a message queue on a remote cluster site. Consequently, processes that use message queues cannot be migrated to other sites.

Return Value

Upon successful completion, a value of 0 is returned and the following actions are taken with respect to the data structure associated with the **msqid** parameter:

msg_qnum is incremented by 1.
msg_lspid is set equal to the process ID of the calling process.
msg_stime is set equal to the current time.

If the **msgsnd** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions

The **msgsnd** system call fails and no message is sent if one or more of the following are true:

EINVAL The **msqid** parameter is not a valid message queue identifier.

EACCES Operation permission is denied to the calling process.

EINVAL **mtype** is less than 1.

EAGAIN The message cannot be sent for one of the reasons stated previously, and **msgflg** is set to **IPC_NOWAIT**.

EINVAL The **msgsz** parameter is less than 0 or greater than the system-imposed limit.

EFAULT The **msgp** parameter points to a location outside of the process's allocated address space.

EINTR **msgsnd** received a signal.

EIDRM The message queue identifier specified by **msqid** has been removed from the system.

Related Information

In this book: "msgctl" in topic 1.2.173, "msgget" in topic 1.2.174, "msgrcv" in topic 1.2.178, "msgxrcv" in topic 1.2.181, and "sigaction, sigvec, signal" in topic 1.2.263.

1.2.181 msgxrcv

Purpose
Receives an extended message.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgxrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgxbuf *msgp;
int msgsz, msgflg;
long msgtyp;
```

Description
The **msgxrcv** system call reads a message from the queue specified by the **msqid** parameter and stores it into the extended message receive buffer pointed to by the **msgp** parameter. The current process must have read permission in order to perform this operation. The **msgxbuf** structure is defined in the **sys/msg.h** header file, and it contains the following members:

```
time_t          mtime; /* Time and date message was sent */
suid_t          muid; /* Sender's effective user ID */
suid_t          mgid; /* Sender's effective group ID */
unsigned long   mnid; /* Sender's node ID */
pid_t           mpid; /* Sender's process ID */
long            mtype; /* Message type */
char            mtext[1]; /* Beginning of message text */
```

The **msgsz** parameter specifies the size of **mtext** in bytes. The receive message is truncated to the size specified by the **msgsz** parameter if it is larger than the **msgsz** parameter and **MSG_NOERROR** is true. The truncated part of the message is lost and no indication of the truncation is given to the calling process.

The **msgsz** parameter specifies the size of **mtext** in bytes. The received message is truncated to the size specified by the **msgsz** parameter if it is larger than the size specified by the **msgsz** parameter and if **MSG_NOERROR** is set in **msgflg**. The truncated part of the message is lost and no indication of the truncation is given to the calling process. If the message is longer than **msgsz** bytes and **MSG_NOERROR** is not set, then the **msgrcv** system call fails and sets **errno** to E2BIG.

The **msgtyp** parameter specifies the type of message requested as follows:

If the **msgtyp** parameter is equal to 0, the first message on the queue is received.

If the **msgtyp** parameter is greater than 0, the first message of the type specified by the **msgtyp** parameter is received.

If the **msgtyp** parameter is less than 0, the first message of the lowest type that is less than or equal to the absolute value of the

msgtyp parameter is received.

The **msgflg** parameter is either 0, or is constructed by logically ORing one or more of the following values:

MSG_NOERROR Truncates the message if it is longer than **msgsz** bytes.

IPC_NOWAIT Specifies the action to take if a message of the desired type is not on the queue:

If **IPC_NOWAIT** is set, then the calling process returns a value of -1 and sets **errno** to ENOMSG.

If **IPC_NOWAIT** is not set, then the calling process suspends execution until one of the following occurs:

- A message of the desired type is placed on the queue.
- The message queue identifier specified by the **msqid** parameter is removed from the system. When this occurs, **errno** is set to EIDRM, and a value of -1 is returned.
- The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes in the manner prescribed in "sigaction, sigvec, signal" in topic 1.2.263.

Warning: If the Transparent Computing Facility is installed, a message queue exists only on the cluster site on which the queue was created. There is no provision for accessing a message queue on a remote cluster site. Consequently, processes that use message queues cannot be migrated to other sites.

Return Value
Upon successful completion, **msgxrcv** returns a value equal to the number of bytes actually stored into **mtext**, and the following actions are taken with respect to the data structure associated with the **msqid** parameter:

msg_qnum is decremented by 1.
msg_lrpid is set equal to the process ID of the calling process.
msg_rtime is set equal to the current time.

If the **msgxrcv** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **msgxrcv** system call fails if one or more of the following are true:

EINVAL **msqid** is not a valid message queue identifier.

EACCES Operation permission is denied to the calling process.

EINVAL **msgsz** is less than 0.

E2BIG **mtext** is greater than **msgsz** and **MSG_NOERROR** is not set.

ENOMSG The queue does not contain a message of the desired type and

IPC_NOWAIT is set.

EFAULT The **msgp** parameter points to a location outside of the process's allocated address space.

EINTR **msgxrcv** received a signal.

EIDRM The message queue identifier specified by **msqid** is removed from the system.

Related Information
In this book: "msgctl" in topic 1.2.173, "msgget" in topic 1.2.174, and "msgrcv" in topic 1.2.178.

1.2.182 NCcollate, NCcoluniq, NCeqvmap, _NCxcol, _NLxcol

Purpose
Collates characters for international character support.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <NLchar.h>


int NCcollate (xc)          int NCcoluniq (xc)
NLchar xc;                  NLchar xc;

int _NCxcol (index, src, xstint NCeqvmap (ucval)
int index;                  int ucval;
NLchar **src, **xstr;

int _NLxcol (index, src, xstr)
int index;
unsigned char **src;
NLchar **xstr;
```

Description

Note: In the multibyte environment, the collation routines listed above are provided for backward compatibility. They are front-ends to the **wc_collate**, **wc_coluniq**, **wc_eqvmap**, **_wcxcol**, and **_mbxcol** routines. Avoid using them if you wish to write portable programs.

The **xc** value is that of an extended character (**NLchar**).

AIX supports a user-configurable collating order per process, using the table file indicated by the **LANG** or **LC_COLLATE** environment variable. Collating values increment from 0. The **NCcollate** macro, called with an **NLchar**, returns the collating value. **NCcollate** returns a negative value if extended collation applies to the **NLchar**. If ***extended collation*** applies, either the **NLchar** is translated to a different character or string of characters before collation (***1-to-n collation***), or the **NLchar** is to collate as a unit with one or more following **NLchar**s (***n-to-1 collation***). For example, the **NLchar** for the code point representing "ö" might translate to the string "oe" before (1-to-**n**) collation or two code points representing "Pi" might translate to a unit "&pi." before (**n**-to-1) collation.

When **NCcollate** determines that extended collation is required, **_NCxcol** or **_NLxcol** should be called.

The **_NCxcol** subroutine performs extended collation on the following:

index The negative value returned from **NCcollate** that indicates that extended collation is needed.

src A pointer to a string of **NLchar** type, starting with the **NLchar** following the one that was passed to the **NCcollate** subroutine.

xstr A pointer to a replacement text string.

For 1-to-**n** collation, **_NCxcol** writes the address to **xstr** of a replacement string that is interpolated into the collating operation ahead of the remaining text of **src**.

For **n**-to-1 collation, a NULL value is written into the pointer.

_NCxcol returns -1 if 1-to-**n** collation is required (**xstr** is not NULL). If **n**-to-1 collation is required, **_NCxcol** returns the collating value of the extended collation.

The **NCcoluniq** macro disables extended collation, simply assigning each **NLchar** a unique value and treating it as a unit. **NCcoluniq** returns its unique collating value, a nonnegative integer that does not receive a special interpretation. A context in which **NCcoluniq** might be used is within character ranges in regular expressions.

The **NCeqvmap** macro is a predicate that returns a nonzero value if the corresponding **NLchar** begins an ***equivalence class***, a set of **NLchars** that can be treated as identical in some collating contexts. For example, if any character of an equivalence class is used as the beginning or ending point of a character range, all of the characters in that class are included in the range.

Related Information

The **ctab** command in *AIX Operating System Commands Reference*.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

AIX Guide to Multibyte Character Set (MBCS) Support.

In this book: "wc_collate, wc_coluniq, wc_eqvmap, _wcxcol, _mbxcol, _wcxcolu, _mbxcolu" in topic 1.2.329.

1.2.183 NCctype

Purpose
Classify characters for international character support environments.

Syntax

#include <NLctype.h>

```
int NCisNLchar  (x)          int NCisalnum  (x)
int x;                       int x;

int NCisalpha  (x)           int NCisspace  (x)
int x;                       int x;

int NCisupper  (x)           int NCispunct  (x)
int x;                       int x;

int NCislower  (x)           int NCisprint  (x)
int x;                       int x;

int NCisdigit  (x)           int NCisgraph  (x)
int x;                       int x;

int NCisxdigit  (x)          int NCiscntrl  (x)
int x;                       int x;
                             int NCisshift  (x)
                             int x;
```

Description

Note: In the multibyte environment, the **NCctype** routines are provided for backward compatibility. All the **NCctype** routines listed above are only front-ends to the **ctype** routines (see "ctype" in topic 1.2.55). Avoid using them if you wish to write portable programs. Note that **NCisshift** always returns 0.

Character classification is user-configurable per process, through the table file indicated by the environment variable **LANG** or **LC_COLLATE**.

These macros classify character-coded integer values using information specified by the current **LANG** or **LC_COLLATE** file configuration. The parameter **x** is tested as an **NLchar** (an extended character); each macro is a predicate form returning 0 for false, and a nonzero value for true. The value of **x** is in the domain of any legal **NLchar** in a value range from 0 to **NLCHARMAX-1** inclusive, or a special value of -1. If the value of **x** is not in the domain of the macro, the result is undefined.

The **NCisNLchar** macro is defined on all valid integer values, whereas the other macros are defined only where **NCisNLchar** is true, and on the special value of -1 (EOF). See " stdio" in topic 1.2.283.

When a nonzero value is returned for **x**:

NCisNLchar **x** is a valid **NLchar** with a value between 0 and **NLCHARMAX**-1, inclusive.

NCisalpha **x** is an alphabetical character.

NCisupper **x** is an uppercase alphabetical character.

NCislower **x** is a lowercase letter.

NCisdigit **x** is a decimal digit (0-9).

NCisxdigit **x** is a hexadecimal digit (0-9, A-F (or a-f)).

NCisalnum **x** is an alphanumeric character or digit.

NCisspace **x** is a space, tab, carriage return, new-line, vertical tab, or form-feed character.

NCispunct **x** is a punctuation character (neither a control character nor an alphanumeric character).

NCisprint **x** is a printing character (including the space character).

NCisgraph **x** is a printing character, excluding the space character.

NCiscntrl **x** is an ASCII delete character (0177) or an ordinary ASCII control character other than the four single-shift characters.

NCisshift **x** is one of the four single-shift characters that is used as the first byte of an extended character.

Related Information
In this book: "conv" in topic 1.2.50, "ctype" in topic 1.2.55, "getc, fgetc, getchar, getw, getwc, fgetwc, getwchar" in topic 1.2.91, "NLchar" in topic 1.2.188, " stdio" in topic 1.2.283, and "environment" in topic 2.4.6.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.184 NCstring

Purpose
Performs operations on strings.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <string.h>


NLchar *NCstrcat (xs1, xs2)          NLchar *NCstrchr (xs, x)
NLchar *xs1, *xs2;                   NLchar *xs, x;

NLchar *NCstrncat (xs1, xs2,         NLchar *NCstrrchr (xs, x)
NLchar *xs1, *xs2;                   NLchar *xs, x;
int n;
                                     NLchar *NCstrpbrk (xs1, s2)
int NCstrcmp (xs1, xs2)              NLchar *xs1;
NLchar *xs1, *xs2;                   char *s2;

int NCstrncmp (xs1, xs2, n)          int NCstrspn (xs1, s2)
NLchar *xs1, *xs2;                   NLchar *xs1;
int n;                               unsigned char *s2;

NLchar *NCstrcpy (xs1, xs2)          int NCstrcspn (xs1, s2)
NLchar *xs1, *xs2;                   NLchar *xs1;
                                     unsigned char *s2;
NLchar *NCstrncpy (xs1, xs2, n)
NLchar *xs1, *xs2;                   NLchar *NCstrtok (xs1, s2)
int n;                               NLchar *xs1;
                                     unsigned char *s2;
int NCstrlen (xs)
NLchar *xs;
```

Description

Note: In the multibyte environment, the **NCstring** routines are provided for backward compatibility. These routines are only front-ends to the **wcstring** routines (see "wcstring" in topic 1.2.327). Avoid using them if you wish to write portable programs.

The **NCstring** subroutines copy, compare, and append strings in memory, and determine such things as location, size, and existence of strings in memory. For these subroutines, a string is an array of **NLchar**s, terminated by a null character. The **NCstring** subroutines parallel the **string** subroutines (see "string" in topic 1.2.288), but operate on strings of type **NLchar** rather than on type **char**, except as specifically noted below.

These subroutines require their parameters (except the **s2** parameter) to be explicitly converted to type **NLchar**, so they should be used on input that is to be scanned many times for each time it is converted. Where this performance concern does not apply, the **NLstring** subroutines are easier to use (see "NLstring" in topic 1.2.193).

The **s2** parameter is a string of type **char** containing code point

representations of ASCII characters or extended characters for international character support. This supports the use of a double-quoted string for this parameter in calling programs.

The parameters **xs1**, **xs2** and **s** point to strings of type **NLchar** (arrays of **NLchar**s terminated by a null character). The **s2** parameter points to strings of type **char**.

The subroutines **NCstrcat**, **NCstrncat**, **NCstrcpy**, and **NCstrncpy** all alter **xs1**. They do not check for overflow of the array pointed to by **xs1**. All string movement is performed character by character and starts at the left. Overlapping moves toward the left work as expected, but overlapping moves to the right may give unexpected results. All of these subroutines are declared in the **string.h** header file.

The **NCstrcat** subroutine appends a copy of the string pointed to by the **xs2** parameter to the end of the string pointed to by the **xs1** parameter. The **NCstrcat** subroutine returns a pointer to the null-terminated result.

The **NCstrncat** subroutine copies at most **n NLchar**s of **xs2** to the end of the string pointed to by the **xs1** parameter. Copying stops before **n NLchar**s if a null character is encountered in the **xs2** string. The **NCstrncat** subroutine returns a pointer to the null-terminated result.

The **NCstrcmp** subroutine lexicographically compares the string pointed to by the **xs1** parameter to the string pointed to by the **xs2** parameter. The **NCstrcmp** subroutine returns a value that is:

Less than 0	If **xs1** is less than **xs2**
Equal to 0	If **xs1** is equal to **xs2**
Greater than 0	If **xs1** is greater than **xs2**.

The **NCstrncmp** subroutine makes the same comparison as **NCstrcmp**, but it compares at most **n** pairs of **NLchar**s. Both **NCstrcmp** and **NCstrncmp** use the environment variable **LANG** or **LC_COLLATE** to determine the collating sequence for performing comparisons. (See "NCcollate, NCcoluniq, NCeqvmap, _NCxcol, _NLxcol" in topic 1.2.182 for information on collation for international character support.) Unless a true collating relationship is to be tested for, **strcmp** and **strncmp** can instead be used for equality comparisons. (See "string" in topic 1.2.288) The bytes will match regardless of the **NLchar**s in the string.

The **NCstrcpy** subroutine copies the string pointed to by the **xs2** parameter to the character array pointed to by the **xs1** parameter. Copying stops when the null character is copied. The **NCstrcpy** subroutine returns the value of the **xs1** parameter.

The **NCstrncpy** subroutine copies **n NLchar**s from the string pointed to by the **xs2** parameter to the character array pointed to by the **xs1** parameter. If **xs2** is less than **n NLchar**s long, then **NCstrncpy** pads **xs1** with trailing null characters to fill **n NLchars**. If **xs2** is **n** or more **NLchar**s long, then only the first **n NLchar**s are copied; the result is not terminated with a null character. The **NCstrncpy** subroutine returns the value of the **xs1** parameter.

The **NCstrlen** subroutine returns the number of **NLchar**s in the string pointed to by the **s** parameter, not including the terminating null character.

The **NCstrchr** subroutine returns a pointer to the first occurrence of the

NLchar specified by the **x** parameter in the string pointed to by the **s** parameter. A NULL pointer is returned if the **NLchar** does not occur in the string. The null character that terminates a string is considered to be part of the string.

The **NCstrrchr** subroutine returns a pointer to the last occurrence of the character specified by the **x** parameter in the string pointed to by the **s** parameter. A NULL pointer is returned if the **NLchar** does not occur in the string. The null character that terminates a string is considered to be part of the string.

The **NCstrpbrk** subroutine returns a pointer to the first occurrence in the string pointed to by the **xs1** parameter of any code point from the string pointed to by the **s2** parameter. A NULL pointer is returned if no character matches.

The **NCstrspn** subroutine returns the length of the initial segment of the string pointed to by the **xs1** parameter that consists entirely of code points from the string pointed to by the **s2** parameter.

The **NCstrcspn** subroutine returns the length of the initial segment of the string pointed to by the **xs1** parameter that consists entirely of code points **not** from the string pointed to by the **s2** parameter.

The **NCstrtok** subroutine returns a pointer to an occurrence of a text token in the string pointed to by the **xs1** parameter. The **s2** parameter specifies a set of code points as token delimiters. If the **s1** parameter is anything other than NULL, then the **NCstrtok** subroutine reads the string pointed to by the **xs1** parameter until it finds one of the delimiter code points specified by the **s2** parameter. It then stores a null character into the string, replacing the delimiter code point, and returns a pointer to the first **NLchar** of the text token. The **NCstrtok** subroutine keeps track of its position in the string so that subsequent calls with a NULL **xs1** parameter step through the string. The delimiters specified by the **s2** parameter can be changed for subsequent calls to **NCstrtok**. When no tokens remain in the string pointed to by the **xs1** parameter, the **NCstrtok** subroutine returns a NULL pointer.

Related Information
In this book: "NCcollate, NCcoluniq, NCeqvmap, _NCxcol, _NLxcol" in topic 1.2.182, "NLchar" in topic 1.2.188, "NLstring" in topic 1.2.193, "NLstrtime" in topic 1.2.194, "string" in topic 1.2.288, and "wcstring" in topic 1.2.327.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.185 netctrl

Purpose
Allows system administrator to exercise control over TCF networking.

Syntax

```
#include <sys/netctrl.h>

int netctrl(option, param)
int option;
caddr_t param;
```

Description
The **netctrl** system call provides the superuser with a method for controlling the behavior of the TCF network (that is, cluster communication). The **option** parameter may be ORed with **SET_OPT** to cause the value(s) to be set or reset as appropriate in addition to returning the old value(s). The various structures to which **param** may refer are defined in **<sys/netctrl.h>**. The following **options** are available:

NET_FLOW Controls TCF cluster communication (network traffic) and/or returns current state. **param** is the address of an integer whose value is **NET_START**, **NETJOIN**, or **NET_STOP**.

- **NET_START** starts a cluster communication, allowing it to join a cluster with other TCF sites.
- **NET_JOIN** is used after cluster communication is enabled to actively look for other cluster sites and hence join a cluster. NET_JOIN is usually used immediately after NET_START.
- **NET_STOP** stops cluster communication.

Note: On AIX/370, in addition to enabling cluster communication, **NET_START** enables communication using a telecommunications adapter (TCA) for use by AIX Access. **NET_STOP**, however, does *not* disable TCA traffic.

NET_STATS Get current TCF network statistics gathered since last network startup. This information is returned into a **net_stats** structure as pointed to by **param**. If **SET_OPT** is set, then the statistics are reset within the kernel.

For the following options, if more sites exist than are implied by the number of entries specified by the **param** argument, the values for those sites are simply not returned or modified.

SITE_STATS
Get current statistics on TCF sites. This information is returned into a **site_stats** structure as pointed to by **param**. If **SET_OPT** is set, then the statistics are reset within the kernel.

MSG_STATS Get current statistics on message traffic on a per site basis. This information is returned into a **msg_stats** structure as pointed to by param. If **SET_OPT** is set, then the statistics are reset within the kernel.

SITE_PARMS

Get or set current site parameters as determined by the **site_parms** structure which is pointed to by **param**. Values for the **site_parms** structure may have the following values:

sp_route Reserved for future implementation.

sp_timeout Must be greater than or equal to 2.

sp_block_size
Must be a power of 2 greater than or equal to 512 and less than or equal to 16384.

sp_retries Must be greater than 0.

sp_window Must be greater than 0.

sp_s_window Must be greater than or equal to 1.

sp_checksum If 1, checksumming is done. If 0, no checksumming is done.

NET_TOPWAIT
Wait for change in TCF cluster topology. The param parameter is a pointer to a character whose value is one of the topology change status values defined in <sys/topchg.h<:

TP_PARTITION
TP_CLEANUP
TP_MERGE
TP_NEWTOP
TP_RECOVERY
TP_CANCEL
TP_WAIT
TP_STABLE

Usually, the first time **NET_WAIT** is used, the value **TP_STABLE** is passed.

The netctrl system call will wait until the current topology change status is different from the provided value; at which point the system call will return, replacing the character pointed at by param with the then current topology status. Because there may be multiple changes in the topology status before this process gets a chance to run--it is possible that the returned topology change status actually equals the value passed in.

Return Value
Upon successful completion, a value of 0 is returned to the calling process. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **netctrl** system call fails if any of the following is true:

EPERM The **SET_OPT** bit was on and the user was not the superuser.

EINVAL Option is not an one of the choices described above.

EINVAL One or more of the parameters specified in the **site_parms**

structure are outside of the acceptable ranges.

EFAULT **param** pointed to an area which was unwritable.

EBUSY **NET_JOIN** was specified while already in the process of joining a cluster.

EINVAL TCF is not enabled on your system (AIX PS/2 only).

Related Information
In this book: "getsites" in topic 1.2.119.

The **clusterstart, clusterstop**, and **netparams** commands in *AIX Operating System Commands Reference*.

1.2.186 NLcatgets

Purpose
Access a message catalog the first time after **NLcatopen** opens the catalog.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <nl_types.h>

char *NLcatgets (catd, set_num,  msg_num, s)
nl_catd catd;
int set_num, msg_num;
char *s;
```

Description
If **NLcatopen** is used to prepare a message catalog for access, then **NLcatgets** must be used to access the message catalog the first time. A successful call to **NLcatopen** returns **catd**. The set and message to retrieve from the catalog is specified by **set_num** and **msg_num**. The pointer to a default string to return if the call fails is **s**.

Return Value
Upon successful completion, **NLcatgets** returns a pointer to the message string from the message catalog. The data held at this location is overwritten on the next call to **NLcatgets**, so it is up to the user program to copy the data and save it before the next call to **NLcatgets**. **NLcatgets** returns a pointer to the default string **s** upon failure.

1.2.187 NLcatopen

Purpose
Deferred open of a message catalog.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <nl_types.h>

nl_catd NLcatopen (filename)
char *filename;
```

Description
The **NLcatopen** subroutine provides a mechanism that prepares a message catalog to be accessed without the overhead of opening the catalog until the first call to **NLcatgets**, when the catalog is actually accessed.

If **NLcatgets** is not used after a call to **NLcatopen**, undefined results occur since no information regarding the message catalog has been prepared.

There is no **NLcatclose** routine to close a catalog that has been opened by **NLcatopen**. Use **catclose** to close a catalog that is opened by **NLcatopen** and accessed by **NLcatgets**.

Return Value
A catalog descriptor is returned upon successful completion. Otherwise, **(nl_catd)-1** will be returned to indicate any failure. The same rules hold true when opening the catalog specified by **filename** as with the **catopen** routine.

1.2.188 NLchar

Purpose
Handles data type NLchar.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <NLchar.h>

typedef unsigned short NLchaint NCencode (x, c)
                                NLchar *x;
int NCdecode (c, x)             unsigned char *c;
unsigned char *c;
NLchar *x;                      int NCencstr (x, c, len)
                                NLchar *x;
int NCdecstr (c, x, len)        unsigned char *c;
unsigned char *c;               int len;
NLchar *x;
int len;                        int NCenc (x, c)
                                NLchar *x;
int NCdec (c, x)                char *c;
char *c;
NLchar *x;                      int NLisNLcp (c)
                                char *c;
int NCdechr (c)
char *c;                        int NLchrlen (c)
                                char *c;
int NCchrlen (nlchr)
NLchar nlchr;
```

Description

Note: In the multibyte environment, these routines which handle the data type **NLchar** are provided for backward compatibility. They are only front-ends to the **wcstring** and **mbstring** routines (see "wcstring" in topic 1.2.327 and "mbstring" in topic 1.2.164). Avoid using them if you wish to write portable programs. There is no **wcstring** equivalent for **NCchrlen**, and there is no **mbstring** equivalent for **NLisNLcp**.

Characters for international character support can be either one or two bytes in length, while all ASCII characters are one byte long. The **NLchar** data type represents both ASCII and extended characters as single units of storage. The **NLchar** subroutines and macros listed here convert between character types **char** and **NLchar** and provide information about a given character of either type.

The **NCdecode** subroutine converts a character starting at **c** into an **NLchar** at **x**, and returns the number of bytes read from **c**. The **NCencode** subroutine makes the inverse translation from type **NLchar** to type **char** and returns the number of bytes written to **c**.

The **NCdecstr** subroutine converts a string of characters from type **char** to type **NLchar**, and the **NCencstr** does the reverse translation. Both subroutines require the address of the source and destination strings and

the total number of elements available for the destination string. The destination string terminates with a zero (0) element, which is included in the string length. The destination length should include space for the terminator. If insufficient space is left for the destination string, a portion of it is not converted and the destination string is not terminated with a 0 byte. The subroutines return the length of the string in elements, including the terminating 0.

The **NCdec** and **NCenc** macros are equivalent to **NCdecode** and **NCencode** respectively. You can use them to avoid the overhead of function calls in situations where the parameters have no side effects.

The **NCdechr** macro is like **NCdecode** except that **NCdechr** simply returns the value of **NLchar** rather than writing the **NLchar** into memory.

The **NLisNLcp**, **NCchrlen**, and **NLchrlen** macros return information about a given character. **NLisNLcp** returns a 0 if the character at **c** is not an extended character, but returns the length of the character if it is an extended character. **NCchrlen** returns the length in bytes that an **NLchar** would have if it were converted into an extended or an ASCII character by **NCencode**. **NLchrlen** returns the length in bytes of the extended or ASCII character starting at **c**.

Related Information
In this book: "conv" in topic 1.2.50, "ctype" in topic 1.2.55, "NCctype" in topic 1.2.183, "wcstring" in topic 1.2.327, and "mbstring" in topic 1.2.164.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

The **axeb**, **ebxa**, and **genxlt** commands in *AIX Operating System Commands*.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.189 NLescstr, NLunescstr, NLflatstr

Purpose
Translates strings of characters.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <NLctype.h>


int NLescstr (src, dest, dleint NLunescstr (src, dest, dlen)
unsigned char *src, *dest;  unsigned char *src, *dest;
int dlen;                   int dlen;
int NLflatstr (src, dest, dlen)
unsigned char *src, *dest;
int dlen
```

Description

Note: In the multibyte environment, these routines are provided for backward compatibility and they support only code page pc850.

These subroutines use the subroutines described under "conv" in topic 1.2.50 to convert an entire string of type **char**, perhaps containing extended characters, into a string of pure ASCII bytes. Each of these subroutines require three parameters: the **src** address of the source string, the **dest** address of the destination string, and the **dlen** value, giving the total number of bytes available in the destination string. Each writes a result string terminated by a null character and returns its length in bytes. The **dlen** value should include space for the null character. If **dest** is too short to contain the entire output string, not all of **src** is translated.

The **NLescstr** uses the **NCesc** subroutine to translate each ASCII or extended character in **src** to pure ASCII. Each extended character encountered is translated to a printable ASCII escape sequence that uniquely identifies the extended character. See "display symbols" in topic 2.4.4 for a list of these escape sequences.

The **NLunescstr** subroutine performs the inverse translation using the **NCunesc** subroutine to translate each ASCII byte of **src** into **dest**, and translate each ASCII escape sequence back into the extended character it represents.

The **NLflatstr** subroutine uses the **NCflatchr** subroutine to translate each character, ASCII or extended, in **src** to a single ASCII byte in **dest**. The **dest** string may have fewer bytes than the **src** string, but the number of logical characters, or the ***display length***, is the same. See "NLstring" in topic 1.2.193.

Related Information
In this book: "ctype" in topic 1.2.55, "getc, fgetc, getchar, getw, getwc, fgetwc, getwchar" in topic 1.2.91, "NCctype" in topic 1.2.183, "NCstring" in topic 1.2.184, "NLchar" in topic 1.2.188, "NLstring" in topic 1.2.193, and "display symbols" in topic 2.4.4.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.190 NLgetctab

Purpose
Finds and maps character collating and classification tables to code points.

Library
Standard C Library (**libc.a**)

Syntax

```
void NLgetctab (ctfile)
char *ctfile;
```

Description
In the multibyte environment, the **NLgetctab** subroutine is a front-end to **setlocale**. If the variable **NLCTAB** is not defined and the parameter passed to it is a NULL pointer, then **NLgetctab** sets the locale to the default **C** locale; otherwise, **NLgetctab** passes the parameter to **setlocale** and behaves as if **setlocale** (**LC_ALL, ctfile**) were executed (see "setlocale" in topic 1.2.251).

Return Value
When **NLgetctab** succeeds, 0 is returned, otherwise -1 is returned.

Related Information
In this book: "NCcollate, NCcoluniq, NCeqvmap, _NCxcol, _NLxcol" in topic 1.2.182, "environment" in topic 2.4.6, and "setlocale" in topic 1.2.251.

The **ctab** command in *AIX Operating System Commands Reference*.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.191 NLgetfile

Purpose
Gets parameter file for international character support.

Library
Standard C Library (**libc.a**)

Syntax

```
int NLgetfile (filename)
char *filename;
```

Description
In the multibyte environment, the **NLgetfile** subroutine is a front-end to **setlocale**. If the variable **NLFILE** is not defined and the parameter is passed to it as a NULL pointer, **NLgetfile** sets the locale to the default **C** locale; otherwise, **NLgetfile** passes the parameter to **setlocale** and behaves as if **setlocale** (**LC_ALL**, **filename**) were called (see "setlocale" in topic 1.2.251).

Return Value
When **NLgetfile** succeeds, 0 is returned.

When **NLgetfile** does not succeed, -1 is returned.

Related Information
In this book: "getenv, NLgetenv" in topic 1.2.94, "NLgetctab" in topic 1.2.190, "setlocale" in topic 1.2.251, and "environment" in topic 2.4.6.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.192 nlist

Purpose
Gets entries from a name list.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <nlist.h>

int nlist (filename, nl)
char *filename;
struct nlist *nl;
```

Description
The **nlist** subroutine allows a program to examines the name list in the executable file named by the **filename** parameter. It selectively extracts a list of values and places them in the array of **nlist** structures pointed to by the **nl** parameter.

The name list specified by the **nl** parameter consists of an array of structures containing names of variables, types, and values. The list is terminated with an element that has a null string in the **name** structure member. Each variable name is looked up in the name list of the file. If the name is found, the **type** and **value** of the name are inserted in the next two fields. The **type** field is set to 0 unless the file was compiled with the **-g** option. If the name is not found, both the **type** and **value** entries are set to 0.

All entries are set to 0 if the specified file cannot be read or if it does not contain a valid name list.

You can use the **nlist** subroutine to examine the system name list kept in the **/unix** file. By examining this list, you can ensure that your programs obtain current system addresses.

The **nlist.h** header file is automatically included by **a.out.h** for compatibility. However, do not include **a.out.h** if you only need the information necessary to use the **nlist** subroutine. If you do include **a.out.h**, follow the **#include** statement with the line:

```
#undef n_name
```

Return Value
Upon successful completion, a value of 0 is returned. If the **nlist** subroutine fails, a value of -1 is returned.

Related Information
In this book: "a.out" in topic 2.3.2.

The **cc** command in *AIX Operating System Commands Reference*.

1.2.193 NLstring

Purpose
Performs operations on strings containing code points.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <string.h>


                unsigned char *NLstrcat (s1,int)NLstrdlen (s)
                unsigned char *s1, *s2;     unsigned char *s;

                unsigned char *NLstrncat (s1unsigned char *NLstrchr (s, x)
                unsigned char *s1, *s2;     unsigned char *s
                int n;                      NLchar x;

                int NLstrcmp (s1, s2)       unsigned char *NLstrrchr (s, x)
                char *s1, *s2;              unsigned char *s
                                            NLchar x;
                int NLstrncmp (s1, s2, n)
                char *s1, *s2;              unsigned char *NLstrpbrk (s1, s2)
                int n;                      unsigned char *s1, *s2;

                char *NLstrcpy (s1, s2)     int NLstrspn (s1, s2)
                char *s1, *s2;              unsigned char *s1, *s2;

                unsigned char *NLstrncpy (s1int2NLstrcspn (s1, s2)
                unsigned char *s1, *s2;     unsigned char *s1,
                int n;                      char *s2;

                int NLstrlen (s)            char *NLstrtok (s1, s2)
                unsigned char *s;           unsigned char *s1, *s2;
```

Description

Note: In the multibyte environment, the **NLstring** routines listed above are provided for backward compatibility. These routines are only front-ends to the **mbstring** routines, (see "mbstring" in topic 1.2.164) and should be avoided if you need to write portable programs.

The **NLstring** subroutines copy, compare, and append strings in memory, and determine such things as location, size, and existence of strings in memory. A string is an array of code points terminated by a NULL character. The **NLstring** subroutines parallel the **string** subroutines (see "string" in topic 1.2.288), and **NLstrcat**, **NLstrncat**, **NLstrcpy**, **NLstrncpy**, and **NLstrlen** are identical in function to their **string** counterparts.

The subroutines **NLstrcat**, **NLstrncat**, **NLstrcpy**, and **NLstrncpy** all alter **s1**. They do not check for overflow of the array pointed to by **s1**. All string movement is performed character by character and starts at the left. Overlapping moves toward the left work as expected, but overlapping moves to the right may give unexpected results. All of these subroutines are declared in the **string.h** header file.

The **NLstrcat** subroutine appends a copy of the string pointed to by the **s2** parameter to the end of the string pointed to by the **s1** parameter. The **NLstrcat** subroutine returns a pointer to the NULL-terminated result.

The **NLstrncat** subroutine performs the same function as the **NLstrcat** subroutine, but the number of bytes appended to the end of the string pointed to by the **s1** parameter is limited to **n**; this may represent fewer than **n** code points. The **NLstrncat** subroutine returns a pointer to the NULL-terminated result.

The **NLstrcmp** subroutine lexicographically compares the string pointed to by the **s1** parameter to the string pointed to by the **s2** parameter. The **NLstrcmp** subroutine returns a value that is:

Less than 0	If **s1** is less than **s2**
Equal to 0	If **s1** is equal to **s2**
Greater than 0	If **s1** is greater than **s2**.

The **NLstrncmp** subroutine makes the same comparison as **NLstrcmp**, but it compares at most **n** bytes. Characters that have 2-byte representations can cause **NLstrncmp** to return 0 for unequal strings. If **n** divides a 2-byte character, then the last byte comparison is skipped. If the only difference in the two strings is in that last byte, an incorrect true is returned.

Both the **NLstrcmp** and **NLstrncmp** subroutines use the environment variable **NLCTAB** to determine the collating sequence for performing comparisons. (See "NCcollate, NCcoluniq, NCeqvmap, _NCxcol, _NLxcol" in topic 1.2.182 for information on collation for international character support.) Unless a true collating relationship is to be tested for, **strcmp** and **strncmp** can instead be used for equality comparisons (see "string" in topic 1.2.288). The bytes will match regardless of code point representations.

The **NLstrcpy** subroutine copies the string pointed to by the **s2** parameter to the character array pointed to by the **s1** parameter. There must be enough room in the array pointed to by the **s1** parameter for the string pointed to by the **s2** parameter, including the trailing NULL character. The **NLstrcpy** subroutine returns the value of the **s1** parameter.

The **NLstrncpy** subroutine copies the string pointed to by the **s2** parameter to the character array pointed to by the **s1** parameter, copying at most **n** bytes. If **s2** is shorter than **n**, a NULL character is added to **s1**. If the length in bytes of **s2** is greater than **n**, the result is not NULL-terminated. If byte **n** is the first byte of an extended code then byte **n** is not copied; **s1** is **n**-1 in length. The **NLstrncpy** subroutine returns the value of the **s1** parameter.

The **NLstrlen** subroutine returns the number of bytes in the string pointed to by the **s** parameter, not including the terminating NULL character.

The **NLstrdlen** subroutine returns the number of code points in the string pointed to by **s**, not including the terminating NULL character.

The **NLstrchr** subroutine returns a pointer to the first occurrence of the code point corresponding to the **NLchar** specified by the **x** parameter in the string pointed to by the **s** parameter. A NULL pointer is returned if the code point does not occur in the string. The NULL character that terminates a string is considered to be part of the string.

The **NLstrrchr** subroutine returns a pointer to the last occurrence of the

code point corresponding to the **NLchar** specified by the **x** parameter in the string pointed to by the **s** parameter. A NULL pointer is returned if the code point does not occur in the string. The NULL character that terminates a string is considered to be part of the string.

The **NLstrpbrk** subroutine returns a pointer to the first occurrence in the string pointed to by the **s1** parameter of any code point from the string pointed to by the **s2** parameter. A NULL pointer is returned if no character matches.

The **NLstrspn** subroutine returns the length of the initial segment of the string pointed to by the **s1** parameter that consists entirely of code points from the string pointed to by the **s2** parameter.

The **NLstrcspn** subroutine returns the length of the initial segment of the string pointed to by the **s1** parameter that consists entirely of code points **not** from the string pointed to by the **s2** parameter.

The **NLstrtok** subroutine returns a pointer to an occurrence of text tokens in the string pointed to by the **s1** parameter. The **s2** parameter specifies a set of code points as token delimiters. If the **s1** parameter is anything other than NULL, then the **NLstrtok** subroutine reads the string pointed to by the **s1** parameter until it finds one of the delimiter code points specified by the **s2** parameter. It then stores a NULL character into the string, replacing the delimiter code point, and returns a pointer to the first code point of the text token. The **NLstrtok** subroutine keeps track of its position in the string so that subsequent calls with a NULL **s1** parameter step through the string. The delimiters specified by the **s2** parameter can be changed for subsequent calls to **NLstrtok**. When no tokens remain in the string pointed to by the **s1** parameter, the **NLstrtok** subroutine returns a NULL pointer.

Related Information
In this book: "NCcollate, NCcoluniq, NCeqvmap, _NCxcol, _NLxcol" in topic 1.2.182, "NCstring" in topic 1.2.184, "NLchar" in topic 1.2.188, "mbstring" in topic 1.2.164, and "string" in topic 1.2.288.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.194 NLstrtime

Purpose
Formats time and date.

Library
Standard C Library (**libc.a**)

Syntax

```
char *NLstrtime (str, len, format, tmdate)
char *str, *format;
int len;
struct tm *tmdate;
```

Description
The **NLstrtime** subroutine converts the internal time and date specification **tmdate** that is generated by the **localtime** or **gmtime** clock structures of **ctime** (see "ctime, localtime, gmtime, asctime, tzset" in topic 1.2.54) into a character string under the direction of **format**. The resulting string is similar to the result of **printf format**, and is placed in the memory location addressed by **str**. It has a maximum length of **len** and terminates with a NULL.

Many conversion specifications are the same as those used by the **date** command. The interpretation of some conversion specifications is affected by the values of environment variables for international character support (see "environment" in topic 2.4.6).

The **format** parameter is a character string containing two types of objects: plain characters that are simply placed in the output string, and conversion specifications that convert information from **tmdate** into readable form in the output string. Each conversion specification is a sequence of this form:

%[[-]width][.precision]type

A **%** (percent sign) introduces a conversion specification.

An optional decimal digit string specifies a minimum field **width**. A converted value that has fewer characters than the field width is padded with spaces to the right. If the decimal digit string is preceded by a minus sign, padding with spaces occurs to the left of the converted value.

If no **width** is given, for numeric fields the appropriate default width is used with the field padded on the left with zeros as required. For strings, the output field is made exactly wide enough to contain the string.

An optional **precision** value gives the maximum number of characters to be printed for the conversion specification. The precision value is a decimal digit string preceded by a period. If the value to be output is longer than the precision, it is truncated on the right.

The **type** of conversion is specified by one or two conversion characters. The characters and their meanings are:

m The month of the year is output as a number between 01 and 12.

h The short month is output as a string established by the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

lh The long month is output as a string established by corresponding entry in the current environment file or the **C** locale if applicable (see "setlocale" in topic 1.2.251.

d The day of the month is output as a number between 01 and 31.

j The Julian day of the year is output as a number between 001 and 366.

w The day of the week is output as a number between 0 and 6.

a The short day of the week is output as a string according to the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

la The long day of the week is output according to the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

y The year is output as a number between 00 and 99.

Y The year is output as a number between 0000 and 9999.

D The date is output in the format specified by the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

lD The long date is output in the format specified by the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

sD The short date is output in the format specified by the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

H The hour of the day is output as a number between 00 and 23.

sH The hour of the day is output as a number between 01 and 12.

M The minute is output as a number between 00 and 59.

S The second is output as a number between 00 and 59.

p The AM or PM indicator is output as a string specified by corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

z The (standard or daylight-saving) time zone name is output as a string from the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

r The time is output in the format specified by the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

T The time is output in the format specified by the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

sT The time is output in the format specified by the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

n Only a new-line character is output.

t Only a tab character is output.

x Nothing is output; this conversion specification is used only as a delimiter.

% The % (percent) character is output.

Related Information
In this book: "NLtmtime" in topic 1.2.195, "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf" in topic 1.2.208, "setlocale" in topic 1.2.251, and "environment" in topic 2.4.6.

The **date** command in *AIX Operating System Commands Reference*.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.195 NLtmtime

Purpose
Sets a time structure from string data.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <time.h>

int NLtmtime (str, format, ptm)
unsigned char *str, *format;
struct tm *ptm;
```

Description
The **NLtmtime** subroutine sets the fields in the **ptm** time structure with information in a **str** text string that is parsed according to the **format** string. For each field descriptor in the **format** string, data is read from the **str** string and placed into appropriate fields of the **ptm** structure. The **format** string is described by these rules:

Each field descriptor begins with a % (percent sign).
A mnemonic string of 1 or 2 characters follows the % sign and indicates the type of field or fields being read.
A blank character (tab, space, or new-line character) anywhere in th **format** string causes all blank characters at the corresponding location in the **str** string to be skipped.
Any character in the **format** string that appears outside of a field descriptor, other than the blank character, must be matched exactly by the same character in the **str** string. If a mismatch occurs, **NLtmtime** stops processing and any information following the mismatch is ignored. The characters and their meanings are:

m The month of the year is input as a number between 01 and 12.

h The short month is input as a string established by the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

lh The long month is input as a string established by the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

d The day of the month is input as a number between 01 and 31.

j The Julian day of the year is input as a number between 001 and 366.

w The day of the week is input as a number between 0 and 6.

a The short day of the week is input as a string according to the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

la The long day of the week is input according to the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

y The year is input as a number between 00 and 99.

Y The year is input as a number between 0000 and 9999.

D The date is input in the format specified by the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

lD The long date is input in the format specified by the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

sD The short date is input in the format specified by the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

H The hour of the day is input as a number between 00 and 23.

sH The hour of the day is input as a number between 01 and 12.

M The minute is input as a number between 00 and 59.

S The second is input as a number between 00 and 59.

p The AM or PM indicator is input as a string specified by corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

z The (standard or daylight-saving) time zone name is input as a string from the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

r The time is input in the format specified by the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

T The time is input in the format specified by the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

sT The time is input in the format specified by the corresponding entry in the current environment file or the **C** locale, if applicable (see "setlocale" in topic 1.2.251).

The field descriptors are the same as those used by **NLstrtime** except for those that do not specify information.

Related Information
In this book: "ctime, localtime, gmtime, asctime, tzset" in topic 1.2.54, "NLstrtime" in topic 1.2.194, "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf, wsscanf" in topic 1.2.241, "setlocale" in topic 1.2.251, and "environment" in topic 2.4.6.

The **date** command in *AIX Operating System Commands Reference*.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.196 NLxin

Purpose
Performs EBCDIC-to-ASCII translation.

Library
Standard C Library (**libc.a**)

Syntax

```
int
NLxin (s1, s2, n)
char *s1, *s2;
int n;
```

Description

The **NLxin** routine performs EBCDIC-to-ASCII translation based on the translation table named by the environment variable **NLIN**. If **NLIN** is not defined or is invalid, **NLxin** uses the default universal EBCDIC-to-ASCII translation table.

s1 is a pointer to an output buffer used to store the translated ASCII data.

s2 is a pointer to the null-terminated EBCDIC character data to be translated.

n is the count of the number of bytes available in *s1*.

NLxin uses the value of the environment variable **NLIN** as a path name to the EBCDIC-to-ASCII translation table. (The inspection of the environment variable NLIN is performed only at the first invocation of **NLxin** within a process. Subsequent invocations of **NLxin** from the same process will use the translation table obtained at the first invocation.)

The byte values from the array *s2* are used to index into the translation table to obtain the ASCII byte that is placed into the character array pointed to by *s1*. The translation proceeds on a byte-by-byte basis until a null-byte is encountered in the array pointed to by *s2*, or *n* bytes have been placed in the array pointed to by *s1*.

Return Value
NLxin returns the number of bytes placed in *s1*.

Related Information
The **axeb**, **ebxa**, and **genxlt** commands in *AIX Operating System Commands Reference*.

The discussion on environment variables in *AIX Operating System Programming Tools and Interfaces*.

1.2.197 NLxout

Purpose
Performs ASCII-to-EBCDIC translation.

Library
Standard C Library (**libc.a**)

Syntax

```
int
NLxout (s1, s2, n)
char *s1, *s2;
int n;
```

Description

The **NLxout** routine performs ASCII-to-EBCDIC translation based on the translation table named by the environment variable **NLOUT**. If **NLOUT** is not defined or is invalid, **NLxout** uses the default universal ASCII-to-EBCDIC translation table.

s1 is a pointer to an output buffer used to store the translated EBCDIC data.

s2 is a pointer to the null-terminated ASCII character data to be translated.

n is the count of the number of bytes available in *s1*.

NLxout uses the value of the environment variable **NLOUT** as a path name to the ASCII-to-EBCDIC translation table. (The inspection of the environment variable NLOUT is performed only at the first invocation of **NLxout** within a process. Subsequent invocations of **NLxout** from the same process will use the translation table obtained at the first invocation.)

The byte values from the array *s2* are used to index into the translation table to obtain the EBCDIC byte that is placed into the character array pointed to by *s1*. The translation proceeds on a byte-by-byte basis until a null-byte is encountered in the array pointed to by *s2*, or *n* bytes have been placed in the array pointed to by *s1*.

Return Value
NLxout returns the number of bytes placed in *s1*.

Related Information
The **axeb**, **ebxa**, and **genxlt** commands in *AIX Operating System Commands Reference*.

The discussion on environment variables in *AIX Operating System Programming Tools and Interfaces*.

1.2.198 nl_langinfo

Purpose
Returns a pointer to a string containing language information.

Syntax

```
#include <langinfo.h>
#include <locale.h>
#include <stdio.h>

char *nl_langinfo (var_id)
int var_id;
```

Description

The **nl_langinfo** subroutine returns a pointer to a string containing information about the particular language or cultural area defined in the program's locale. The manifest constant names and values of items are defined in **langinfo.h**.

The following items of cultural data are defined in this locale:

Item	Category	Setting
D_T_FMT	LC_TIME	"%a%b%d%H:%M:%S%Y"
D_FMT	LC_TIME	"%m/%d/%y"
T_FMT	LC_TIME	"%H:%M:%S"
AM_STR	LC_TIME	"AM"
PM_STR	LC_TIME	"PM"
DAY_1	LC_TIME	"Sunday"
DAY_2	LC_TIME	"Monday"
DAY_3	LC_TIME	"Tuesday"
DAY_4	LC_TIME	"Wednesday"
DAY_5	LC_TIME	"Thursday"
DAY_6	LC_TIME	"Friday"
DAY_7	LC_TIME	"Saturday"
ABDAY_1	LC_TIME	"Sun"
ABDAY_2	LC_TIME	"Mon"
ABDAY_3	LC_TIME	"Tue"
ABDAY_4	LC_TIME	"Wed"
ABDAY_5	LC_TIME	"Thu"

ABDAY_6	LC_TIME	"Fri"
ABDAY_7	LC_TIME	"Sat"
MON_1	LC_TIME	"January"
MON_2	LC_TIME	"February"
MON_3	LC_TIME	"March"
MON_4	LC_TIME	"April"
MON_5	LC_TIME	"May"
MON_6	LC_TIME	"June"
MON_7	LC_TIME	"July"
MON_8	LC_TIME	"August"
MON_9	LC_TIME	"September"
MON_10	LC_TIME	"October"
MON_11	LC_TIME	"November"
MON_12	LC_TIME	"December"
ABMON_1	LC_TIME	"Jan"
ABMON_2	LC_TIME	"Feb"
ABMON_3	LC_TIME	"Mar"
ABMON_4	LC_TIME	"Apr"
ABMON_5	LC_TIME	"May"
ABMON_6	LC_TIME	"Jun"
ABMON_7	LC_TIME	"Jul"
ABMON_8	LC_TIME	"Aug"
ABMON_9	LC_TIME	"Sep"
ABMON_10	LC_TIME	"Oct"
ABMON_11	LC_TIME	"Nov"
ABMON_12	LC_TIME	"Dec"
RADIXCHAR	LC_NUMERIC	"."
THOUSEP	LC_NUMERIC	""
YESSTR	LC_ALL	"yes"
NOSTR	LC_ALL	"no"

CRNCYSTR	LC_MONETARY	""

Return Value
If the **setlocale** subroutine has not initialized the program's locale, the **nl_langinfo** subroutine returns a pointer to the corresponding string in the **C** locale. In all locales, the **nl_langinfo** subroutine returns a pointer to an empty string if an item contains an invalid setting.

Related Information
In this book: "setlocale" in topic 1.2.251, "langinfo.h" in topic 2.4.10, and "nl_types.h" in topic 2.4.19.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.199 open, openx, creat

Purpose
Opens a file for reading or writing.

Syntax

```
#include <fcntl.h>


int open (path, oflag, mode)int creat (path, mode)
char *path;                  char *path;
int oflag, mode;             int mode;
int openx (path, oflag, mode, ext)
char *path;
int oflag, mode, ext;
```

Description
The **open** and **openx** system calls open a file descriptor for the file named by the **path** parameter.

The file status flags are set according to the value of the **oflag** parameter. The **oflag** parameter values are constructed by logically ORing flags from the following list:

Note: Do not use **O_RDONLY, O_WRONLY,** or **O_RDWR** together.

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_DEFERC Open with defer commit update semantics. This is used with other modes indicating the file is open for modification. With this mode, changes to the file are not made permanent until either an **fcommit**, **fsync**, **close**, or **exit** are done (or the program aborts). Changes made since the last commit can be undone using **fabort**. These semantics may not apply if the file is open for modification multiple times and all opens are not with the **O_DEFERC** flag set. See also "fsync, fcommit" in topic 1.2.87, "fabort" in topic 1.2.75, and "close, closex" in topic 1.2.48.

O_NDELAY Open with no delay. This flag may affect subsequent reads and writes.

When a FIFO is opened with **O_RDONLY** or **O_WRONLY** set, the following facts apply:

If **O_NDELAY** is set, an **open** for reading-only returns without delay. An **open** for writing-only returns an error if no process currently has the file open for reading.

If **O_NDELAY** is clear, an **open** for reading-only blocks until a process opens the file for writing. An **open** for writing-only blocks until a process opens the file for reading.

When opening a file associated with a communication line:

If **O_NDELAY** is set, the **open** returns without waiting for carrier.

If **O_NDELAY** is clear, the **open** blocks until carrier is present.

When opening a regular file that supports enforced record locks:

If **O_NDELAY** is set, then reads and writes to portions of the file that are locked by other processes return an error.

If **O_NDELAY** is clear, then reads and writes to portions of the file that are locked by other processes blocks until the locks are released.

O_NONBLOCK Open with no delay. This flag is identical in function to **O_NDELAY** when opening a file. However, subsequent reads and writes return different values based on whether the file is opened with **O_NDELAY** or **O_NONBLOCK**. See "read, readv, readx" in topic 1.2.224 and "write, writex" in topic 1.2.330.

O_APPEND If set, the file pointer is set to the end of the file prior to each write.

O_CREAT If the file exists, this flag has no effect. If the file does not exist, the file is created. The file's owner ID is set to the process's effective user ID. If the **S_ISGID** mode bit is set in the parent directory, the file's group ID is set to the group ID of the parent directory. Otherwise, the file's group ID is set to the process's effective group ID. The low-order 12 bits of the file mode are set to the value of the **mode** parameter modified as follows:

All bits set in the process's file mode creation mask are cleared. (For information about the creation mask, see "umask" in topic 1.2.314.)

The **S_ISVTX** bit of the mode, which saves the text image after execution, is cleared.

For information about file modes and a list of the mode values, see "chmod, fchmod" in topic 1.2.44 and "stat.h" in topic 2.4.22.

O_TRUNC If the file exists, then its length is truncated to 0, and the mode and owner are unchanged. This requires write permission on the file. If the file has any outstanding record locks, then **open** fails and the file remains unchanged. The **S_ISGID** and **S_ISUID** mode bits are cleared.

O_EXCL If **O_EXCL** and **O_CREAT** are set, **open** fails if the file exists.

O_SYNC Open with immediate commit update semantics. This is used with other modes indicating the file is open for modification. With this mode, changes to the file are guaranteed to be made permanent (with an implicit **fsync**) before the **write** system call to be considered atomic, performing both write and commit functions before returning. This flag is only effective with regular and block special files.

O_REPLSYNC Open with immediate replicated commit update semantics. This is only meaningful when used with the **O_SYNC** flag and if you have the TCF (Transparent Computing Facility) installed. In the case where a file is replicated on more than one site, this flag guarantees that the modified file will be sorted on at least two file systems before the **write** system call completes. These two file systems necessarily will be the primary and either a backbone or secondary sites.

This flag will be ignored when either the **O_SYNC** bit is not set, the file is not replicated, or no backbone site is available in the cluster to replicate the file. When writing data to this file with the **write** system call, errors reported through the return code reflect only the success or failure of the primary copy of the file.

Propagation errors which include not having a backbone site for the file in the current cluster or having a backbone file system run out of free space are noted in the system log (see **syslog** in the *AIX Operating Systems Commands Reference*). Thus, no change in **write** return status is noted; the system log must be queried to ascertain possible errors. Finally, this flag is only effective with regular files.

The file pointer used to mark the current position within the file is set to the beginning of the file. The **mode** parameter is used only if **open** or **openx** creates the file named path (as requested by **O_CREAT**, described above). The **ext** parameter provides communication with character device drivers that require additional information or provide additional status. Each driver interprets the **ext** in a device-dependent way, either as a value or as a pointer to a communication area. Drivers must apply reasonable defaults when the **ext** parameter is 0.

The new file descriptor is set to remain open across **exec** system calls (see "fcntl, flock, lockf" in topic 1.2.78).

No process can have more than 200 file descriptors open simultaneously.

If the named file is a hidden directory, (see "chhidden" in topic 1.2.42) **open** or **openx** selects one of the existing components inside the hidden directory according to the process's **xvers** string (see "getxvers, setxvers" in topic 1.2.129) and site path list (see "getspath, setspath" in topic 1.2.122). A component within the hidden directory is created by **open** only if **O_CREAT** (see "Hidden Directories" in topic 1.1.5.1.5) is specified and the component is named explicitly, using the @ syntax.

If the Transparent Computing Facility is installed, the following information applies to replicated files and special files.

When creating a file in a replicated file system, or opening an existing file in that file system with **O_WRONLY** or **O_RDWR**, the primary copy is used as the storage site; the site which provides the data for reading and writing. This is because the primary copy of the file system is the only one on which change can be made. The other copies of the file system are brought up-to-date when changes to the file are committed. While a file is open for **O_WRONLY** or **W_RDWR**, all read and writes on the file by any process will be using the same copy of the file.

When all processes which have a replicated file open have it open for **O_RDONLY**, the system may provide data to each process from any up-to-date copy of the file, (that is, several different copies/sites can provide data when only read access is being used by all processes using the files) often a non-primary copy stored on the site where the process is running. If this storage site goes down, files open with **O_RDONLY** will continue to read uninterrupted if another storage site for the file can be found.

If the named file is in a replicated file system, the site that stores the primary copy of the file system must be up. If the file does not previously exist, upon creation it is replicated on only the primary and backbone copies of the file system. Once created, the **chfstore** system call can be used to increase the number of sites on which the file is stored.

TTY devices and block devices can be opened from any site within the TCF cluster. To be able to open a remote TTY device, that device must already be held open by at least one process in the cluster; that is, the first open of a TTY device must be local. The site to which the physical device is attached is identified by the device site specified in **mknod** or **mknodx** when the device special file was created.

Character devices other than TTY devices and null can only be opened at the site which has the device.

Note that the following two system calls are equivalent:

creat (*path, mode*)

open (*path*, **O_WRONLY** | **O_CREAT** | **O_TRUNC**, *mode*)

Upon successful completion, the file descriptor, a nonnegative integer, is returned. If **open** fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **open** system call fails, and the named file is not opened if one or more of the following are true:

ENOTDIR A component of the path prefix is not a directory.

ENOENT **O_CREAT** is not set and the named file does not exist.

EACCES A component of the path prefix denies search permission.

EACCES The type of access specified by the **oflag** parameter is denied for the named file.

EISDIR The named file is a directory and the **oflag** parameter is write or read/write.

EROFS The named file resides on a read-only file system and the **oflag** parameter is write or read/write.

EMFILE Two hundred (200) file descriptors are currently open.

ENXIO The named file is a character special or block special file, and the device associated with this special file does not exist.

ENXIO The named file is a multiplexed special file and either the channel number is outside of the valid range, or no more channels are available.

ETXTBSY The file is a pure procedure (shared text) file that is being executed and the **oflag** parameter is write or read/write.

EFAULT The **path** parameter points to a location outside of the process's allocated address space.

EEXIST **O_CREAT** and **O_EXCL** are set, and the named file exists.

ENXIO **O_NDELAY** is set, the named file is a **FIFO**, **O_WRONLY** is set, and no process has the file open for reading.

EAGAIN **O_TRUNC** is set, and the named file contains a record lock owned by another process. See "fcntl, flock, lockf" in topic 1.2.78 for information about record locks.

EINTR A signal was caught during the **open** system call.

ENFILE The system file table or inode table is full.

ENOSPC The directory that would contain the new file cannot be extended.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

ENAMETOOLONG
A component of the **path** parameter exceeded **NAME_MAX** characters or the entire path parameter exceeded **PATH_MAX** characters.

EIO An I/O error occurred during the operation.

ENOENT **O_CREAT** is set but a directory in the **path** prefix does not exist.

A hidden directory was named, but no component inside it matched the process's current site **path** list.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

ELOOP A loop of symbolic links was detected.

ENOSPC The file system is out of inodes.

EDQUOT The directory in which the entry for the new link is being placed cannot be extended because the user's quota of disk blocks or inodes on the file system containing the directory has been exhausted.

EINVAL An invalid combination of open modes (**O_RDONLY**, **O_WRONLY** and **O_RDWR**) is specified, or the named file is a **FIFO** and **O_RDWR** is set.

ELOCK A named pipe already open for read is opened again for read, or already open for write and is opened again for write.

EOPNOTSUPP
A socket was named, and opens on sockets are not permitted.

If the Transparent Computing Facility is installed on your system, **open** can also fail if one or more of the following are true:

ESITEDN1 **path** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE **path** is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of **path** is replicated but not stored on any site which is currently up.

EROFS Write access is requested for a file on a replicated file system in which the primary copy is unavailable.

ENLDEV The named file is a non-**tty** character special file which corresponds to a device physically attached to another site in the cluster.

EINTR A signal was caught during the system call.

Related Information
In this book: "chmod, fchmod" in topic 1.2.44, "close, closex" in topic 1.2.48, "dup" in topic 1.2.64, "fabort" in topic 1.2.75, "fcntl, flock, lockf" in topic 1.2.78, "fsync, fcommit" in topic 1.2.87, "lseek" in topic 1.2.161, "mknod, mknodx, mkfifo" in topic 1.2.169, "open, openx, creat," "read, readv, readx" in topic 1.2.224, "umask" in topic 1.2.314, "write, writex" in topic 1.2.330, and "stat.h" in topic 2.4.22.

1.2.200 pad: sflip, sflipa, lflip, lflipa, get_howflip, PAD, PADOPEN, PADCLOSE

Purpose
Facilitate the exchange of binary data between processes in a heterogeneous network.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/param.h>

short sflip(s, howflip);
short s;
int howflip;

void sflipa(as, howflip, numshort);
short as[];
int howflip;
int numshort;

long lflip(l, howflip);
long l;
int howflip;

void lflipa(al, howflip, numlong);
long al[]
int howflip;
int numlong;

int get_howflip(CPU_code);
short CPU_code;
```

(The following padding routines are macros, NOT functions.)

```
PAD(decl, uname)

PADOPEN(stag)
PADCLOSE(stag, sname, uname)
```

Description
These macros and procedures facilitate the exchange of binary data between processes in a heterogeneous network. They provide a framework which application programs may use to allow non-ASCII data written by one process to be read by another process, even if the two processes are not executing on the same type of machine.

The need for this mechanism is several-fold. First, different C compilers may have different alignment constraints for fields within a structure. For instance, a C compiler for one CPU type may require that 32-bit fields begin on a 32-bit boundary, while a C compiler for another type of CPU might allow a 32-bit field to begin on an 8-bit boundary. Such a difference could lead to different structure sizes and field offsets for the same structure definition. Programs can avoid these differences by defining the format of their data structures more explicitly. One way is to define their structures using the padding macros defined below. These macros will add padding characters between fields where necessary so that

the alignment constraint of all C compilers is satisfied.

Second, different types of CPUs may have different byte orderings. That is, the order in which 8-bit bytes are layed out inside 16- and 32-bit words varies according to each machine's architecture. Processes which read binary data may use the byte flipping routines declared above to reorder the bytes in 16- and 32-bit values to agree with the byte ordering of its own CPU.

Third, in addition to the unpadded structures described above, certain data types may refer to different size objects when the same definitions are compiled for different machines. This list includes 'int's', pointer types and procedures. Therefore, a structure that includes any of these data types is not portable and should not be used when writing data which is to be read by a process on a different type of CPU.

Finally, floating point values do not have the same representation on all types of CPUs. While it may be possible to write routines to convert between these different formats, no such library routines are provided at this time.

Subtopics
1.2.200.1 PADDING MACROS
1.2.200.2 BYTE FLIPPING ROUTINES

1.2.200.1 PADDING MACROS

The following padding macros should be used inside all structure definitions which define the format of non-character data written by one process which is to be read by another process. If all of data being written is character data (including 8-bit numeric data), no padding is necessary. If some of the data is 16-bit or 32-bit data, all 8-bit and 16-bit data should be padded, and it is recommended that 32-bit data also be padded.

There are two ways to pad fields within a structure. The first uses the **PADOPEN** and **PADCLOSE** macros to surround a collection of like-size fields (each 8, 16, or 32 bits). The second uses the **PAD** macro to pad a single field.

The **PADOPEN/PADCLOSE** method has the advantage of providing a name for the collection of like size fields (for use by the byte flipping routines) and, therefore, it is easy to add a new field at a later time without modification of those byte flipping routines.

The **PAD** macro has the advantage that the size of an object may easily be changed (from 8 bits to 16 bits, for example) since there is no requirement that like size fields appear together.

For example, if you wanted to write the following structure:

```
struct abc {
     long  along;
     long  blong;
     short ashort;
     short bshort[5];
     short cshort;
     char  achar;
     char  bchar[6];
};
```

You could instead use the **PADOPEN/PADCLOSE** macros as follows to generate an equivalent structure with the same field names. Note that the **PADOPEN** and **PADCLOSE** are not followed by semicolons. Also, **NUM_LABC**, **NUM_SABC** and **NUM_CABC** compute the number of long, shorts and chars, respectively. These values will be used by the flipping routines.

```
struct abc {
     PADOPEN(Labc)
     long  _along;
     long  _blong;
     PADCLOSE(Labc, L1abc, L2abc)
#define along L2abc.L1abc._along
#define blong L2abc.L1abc._blong
#define NUM_LABC   (sizeof (struct Labc) / sizeof (long))

     PADOPEN(Sabc)
     short _ashort;
     short _bshort[5];
     short _cshort;
     PADCLOSE(Sabc, S1abc, S2abc)
#define ashort S2abc.S1abc._ashort
#define bshort S2abc.S1abc._bshort
#define cshort S2abc.S1abc._cshort
#define NUM_SABC   (sizeof (struct Sabc) / sizeof (short))
```

```
            PADOPEN(Cabc)
            char  _achar;
            char  _bchar[6];
            PADCLOSE(Cabc, C1abc, C2abc)
       #define achar           C2abc.C1abc._achar
       #define bchar           C2abc.C1abc._bchar
       #define NUM_CABC (sizeof (struct Cabc) / sizeof (char))
       };
```

Alternatively, if you had a simpler structure where each field was no bigger than 32 bits:

```
       struct xyz {
            short ashort;
            char  achar;
            long  along;
       };
```

You could use the **PAD** macro and instead write the following. Again, note that the **PAD** macro is not followed by a semicolon.

```
       struct xyz {
            PAD(short _ashort, xyz1)
            PAD(char  _achar,  xyz2)
            PAD(long  _along,  xyz3)
       };
       #define ashort xyz1._ashort
       #define achar  xyz2._achar
       #define along  xyz3._along
```

Below are the actual declarations for the padding arrays.

```
  #define       ALIGN_SZ         (sizeof(align_t))
  #define       ALIGN_ROUND            (ALIGN_SZ - 1)


  /* PADOPEN  -- begin a padded range of structure fields using tag
   *         'stag'. */

  #define       PADOPEN(stag) \\
        union { struct stag {


  /* PADCLOSE -- end a padded range of structure fields.
   *     'stag' should be the same tag used in the preceding PADOPEN.
   *     'sname' and 'uname' should be unique names.
   */

  #define       PADCLOSE(stag, sname, uname) \\
        }  sname;  \\
        PADARRAY((struct stag), PADNAME) } uname;

  /* PADARRAY -- generate an array declaration of a size such that
   * the union of 'padobj' and this array will meet the heterogeneous
   * structure size criterion.
   */
```

```
#define       PADARRAY(padobj, arrayname) \\
      align_t arrayname[(sizeof padobj + ALIGN_ROUND) / ALIGN_SZ];


/* PAD -- Surround declaration 'decl' with a union that also includes
 * an align_t for padding.
 */

#define       PAD(decl, uname) \\
      union { decl; align_t PADNAME; } uname;
```

1.2.200.2 BYTE FLIPPING ROUTINES

The byte flipping routines should be used by a process that is reading binary data, if that binary data may have been written by a process running on a different type of CPU.

Character data (including 8-bit numeric data) need not be byte flipped. Only 16-bit (short) and 32-bit (long) values need to be flipped to convert from the byte ordering of the writing site to the byte ordering of the reading site.

There are four flipping routines provided:

lflip Flips a long value, returning the result after flipping its argument.

sflip Flips a short value, returning the result after flipping its argument.

lflipa Flips in place an array of longs (or a structure, all of whose elements are longs).

sflipa Flips in place an array of shorts (or a structure, all of whose elements are shorts).

Each of these flipping routines takes a parameter **howflip** which indicates how the bytes are to be reordered. This value may be obtained from the routine **get_howflip**, to which the reading process must provide the CPU type code of the site from which the data was written.

Note: It is assumed in the following that the reader is able to determine the CPU type code for the CPU from which the data was written. This information is needed so that the flipping routines know the byte ordering of the writing site, and therefore how the bytes must be reordered to conform to the byte ordering of the local site. Possible ways that the CPU type code may be determined include:

Explicitly recording this in the text of the application program.

Deriving it from a site number or site name using one of the routines described in **sf**. The site number or site name might, in turn, be determined from the path name used to open the file or from the site on which the file is stored.

Using the same examples as above, consider a process reading files whose formats are described by the structures **abc** and **xyz**. The structure **abc** was declared using the **PADOPEN** and **PADCLOSE** macros, while the structure **xyz** was declared using the **PAD** macro. The reading process could look like this:

```
struct abc abcbuf;
struct xyz xyzbuf;

read(fildes1, &abcbuf, sizeof abcbuf);
flipabc(&abcbuf);

read(fildes2, &xyzbuf, sizeof xyzbuf);
flipxyz(&xyzbuf);
```

Where the byte flipping routines **flipabc** and **flipxyz** might then be defined:

```
flipabc(buf, writer)
    struct abc *buf;
 siteno_t writer;
{
   short    xcode   = sfnum(writer)->sf_xcode;
   int      howflip = get_howflip(xcode);

   lflipa((long *) &(buf->L2abc), howflip, NUM_LABC);
   sflipa((short *) &(buf->S2abc), howflip, NUM_SABC);
}

flipxyz(buf)
   struct xyz *buf;
 short  xcode;
{
   int      howflip = get_howflip(xcode);

   buf->ashort = sflip(buf->ashort, howflip);
   buf->along = lflip(buf->along, howflip);
}
```

Related Information

In this book: "getut: getutent, getutid, getutline, pututline, setutent, endutent, utmpname" in topic 1.2.126.

1.2.201 pathconf, fpathconf

Purpose
Retrieves file implementation characteristics.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>

long pathconf (path, name)        long fpathconf (fildes, name)
char *path;                       int fildes;
int name;                         int name;
```

Description
The **pathconf** system call allows an application to determine the characteristics of operations supported by the file system underlying the file named by **path**. Read, write, or execute permission of the named file is not required, but all directories in the path leading to the file must be searchable.

The **fpathconf** system call allows an application to retrieve the same information for an open file. In this case, **fildes** is a file descriptor returned by a successful **openx, fcntl**, or **pipe** system call.

The **name** parameter specifies the configuration attribute to be queried. If this attribute is not applicable to the file specified by **path** or **fildes**, **pathconf** returns an error. Symbolic values for the **name** parameter are defined in **unistd.h**:

Attribute	Meaning
_PC_LINK_MAX	The maximum number of links to the file; this is usually either 1 or 65,535.
_PC_MAX_CANON	The maximum number of bytes in a canonical input line. This is applicable only to terminal devices.
_PC_MAX_INPUT	The number of types allowed in an input queue. This is applicable only to terminal devices.
_PC_NAME_MAX	Maximum number of bytes in a file name (not including a terminating NULL); this may be as small as 14, but is never larger than 255. This is applicable only to a directory file.
_PC_PATH_MAX	Maximum number of bytes in a path name (not including a terminating NULL); this is never larger than 65,535.
_PC_PIPE_BUF	Maximum number of bytes guaranteed to be written atomically. This is applicable only to a FIFO.
_PC_CHOWN_RESTRICTED	Returns 1 indicating that the **chown** function is restricted to use by the superuser.
_PC_NO_TRUNC	Returns 1 if supplying a component name longer than

allowed by **_PC_PATH_MAX** causes an error. Returns 0 if long component names are truncated. This is applicable only to a directory file.

_PC_VDISABLE This is always 0; no disabling character is defined. This is applicable only to a terminal device.

Return Value
If the **pathconf** or **fpathconf** system call is successful, the specified parameter is returned. If the **pathconf** or **fpathconf** system call fails, a value of -1 is returned and **errno** is set to indicate the error. If the value corresponding to **name** has no limit for the path or file descriptor, the **pathconf** and **fpathconf** system calls return -1 without changing **errno**.

Error Conditions
The **pathconf** system call fails if one or more of the following are true. (These errors are applicable only to the **pathconf** system call and any service which requires path name resolution.)

ENOTDIR A component of the path prefix is not a directory.

ENOENT A component of the path prefix does not exist, or the process has the **disallow truncation** attribute.

EACCES Search permission is denied on a component of the path prefix.

ENOENT The path name is null.

ESTALE The process's root or current directory is located in an NFS virtual file system that has been unmounted.

EFAULT The **path** parameter points to a location outside of the process's allocated address space.

ELOOP Too many symbolic links were encountered in translating the path name.

ENAMETOOLONG
A component of a path name exceeded 255 characters, or an entire path name exceeded 1023 characters.

EIO An I/O error occurred during the operation.

EINTR A signal was caught during the system call.

The **fpathconf** system call fails if the following is true:

EBADF The **fildes** parameter does not refer to an open file.

The **pathconf** and the **fpathconf** system calls fail if the following is true:

EINVAL The **name** parameter specifies an unknown or inapplicable characteristic.

Related Information
In this book: "sysconf" in topic 1.2.296.

1.2.202 pause

Purpose
Suspends a process until a signal is received.

Syntax

int pause ( )

Description
The **pause** system call suspends the calling process until it receives a signal. The signal must not be one that is ignored by the calling process. **pause** does not affect the action taken upon the receipt of a signal.

If the signal received causes the calling process to terminate, then the **pause** system call does not return.

If the signal is caught by the calling process and control is returned from the signal-catching function, then the calling process resumes execution from the point of suspension; the **pause** system call returns a value of -1 and sets **errno** to EINTR. (For information about signal-catching functions, see "sigaction, sigvec, signal" in topic 1.2.263.)

Related Information
In this book: "alarm" in topic 1.2.14, "kill, kill3, killpg" in topic 1.2.138, "sigaction, sigvec, signal" in topic 1.2.263, and "wait, waitpid" in topic 1.2.325.

1.2.203 perror

Purpose
Writes a message explaining a system call error.

Library
Standard C Library (**libc.a**)

Syntax

```
void perror (s)                 extern int errno;
char *s;                        extern char *sys_errlist [ ];
                                extern int sys_nerr;
```

Description
The **perror** subroutine writes a message on the standard error output that describes the last error encountered by a system call or library subroutine. The error message includes the parameter string **s** followed by a **:** (colon), a blank, the message, and a new-line character. To be of the most use, the parameter string **s** should include the name of the program that caused the error. The error number is taken from the external variable **errno**, which is set when an error occurs, but is not cleared when a successful call is made. See Appendix A, "Error Codes" in topic A.0 for a discussion of **errno** values and their meanings.

To simplify various message formats, the array of message strings **sys_errlist** is provided. Use **errno** as an index into this table to get the message string without the new-line character. The largest message number provided in the table is **sys_nerr**. Be sure to check **sys_nerr** because new error codes may be added to the system before they are added to the table.

Related Information
In this book: "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf" in topic 1.2.208.

1.2.204 pipe

Purpose
Creates an interprocess channel.

Syntax

int pipe (fildes)
int fildes[2];

Description
The **pipe** system call creates an interprocess channel called a pipe and returns two file descriptors, **fildes[0**] and **fildes[1**]. The **fildes[0**] file descriptor is opened for reading and **fildes[1**] is opened for writing.

A read on file descriptor **fildes** [**0**] accesses the data written to **fildes[1**] on a first-in, first-out basis.

When writing, at least 40,960 bytes of data are buffered by the pipe before the writing process is blocked.

Warning: The actions of the **pipe** system call are undefined if the **fildes** parameter points to a location outside of the process's allocated address space.

A pipe can be inherited through the **fork**, **rfork** and **run** system calls and held open as a process moves to a different site using the **rexec** or **migrate** system calls. Through combinations of these operations, it is possible to set up one or more processes reading data from the read end of the pipe, and one or more processes writing data to the write end of a pipe. These processes may be on one site, on different sites but of the same CPU type, or on sites with differing CPU types. If the reader and writer are on different sites, slightly better performance results if the **pipe** call is issued at the site of the writer. In any case, correct operation should be independent of the relationships between the sites of the reader, writer, and issuance of the **pipe** system call.

Return Value
Upon successful completion, a value of 0 is returned. If **pipe** fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **pipe** system call fails if one or more the following are true:

EFAULT The **fildes** parameter points to a location outside of the process's allocated address space.

EMFILE 199 or more file descriptors are already open.

ENFILE The system file table or inode table is full.

ENOSPC There is no space to create a new pipe in the pipe file system (there are no more inodes available).

Related Information
In this book: "read, readv, readx" in topic 1.2.224, "select" in topic 1.2.242, and "write, writex" in topic 1.2.330.

The **sh** command in *AIX Operating System Commands Reference*.

1.2.205 plock

Purpose
Locks the process, text, or data in memory.

Syntax

```
#include <sys/lock.h>

int plock (op)
int op;
```

Description
The **plock** system call allows the calling process to lock or unlock its text segment (***text lock***), its data segment (***data lock***), or both its text and data segments (***process lock***) into memory. Locked segments are pinned in memory and are immune to all routine paging. The effective user ID of the calling process must be superuser to use this call.

The **op** parameter specifies one of the following operations:

PROCLOCK Locks text and data segments into memory (process lock).
TXTLOCK Locks text segment into memory (text lock).
DATLOCK Locks data segment into memory (data lock).
UNLOCK Removes locks.

Return Value
Upon successful completion, a value of 0 is returned to the calling process. If **plock** fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **plock** system call fails if one or more of the following are true:

EPERM The effective user ID of the calling process is not superuser.

EINVAL The **op** parameter has a value other than **PROCLOCK**, **TXTLOCK**, **DATLOCK**, or **UNLOCK**.

EINVAL **op** is equal to **PROCLOCK** and a process lock, a text lock, or a data lock already exists on the calling process.

EINVAL **op** is equal to **TXTLOCK** and a text lock, or a process lock already exists on the calling process.

EINVAL **op** is equal to **DATLOCK** and a data lock, or a process lock already exists on the calling process.

EINVAL **op** is equal to **UNLOCK** and no type of lock exists on the calling process.

Related Information
In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, "exit, _exit" in topic 1.2.73, and " fork, vfork" in topic 1.2.83.

1.2.206 plot

Purpose
Performs graphic output.

Library
Graphics Libraries (**libplot.a**, **libprint.a**, **lib300.a**, and others)

Syntax

```
void openpl ( )

void erase ( )

void label (s)
char *s;

void line (x1, y1, x2, y2)
int  x1, y1, x2, y2;

void circle (x, y, r)
int x, y, r;

void arc (x, y, x0, y0, x1,
int x, y, x0, y0, x1, y1;

void move (x, y)
int x, y;

void cont (x, y)
int x, y;

void point (x, y)
int x, y;

void linemod (s)
char *s;

void space (x0, y0, x1, y1)
int x0, y0, x1, y1;

void closepl ( )
```

Description
The **plot** subroutine family generates graphic output in a relatively device-independent manner. The **space** subroutine must be used before any of these functions to declare the amount of space necessary. The **openpl** subroutine must be used before any of the others to open the device for writing. The **closepl** subroutine flushes the output.

The **circle** subroutine draws a circle of radius **r** with center at the point (**x**, **y**). The **arc** subroutine draws an arc of a circle with center at the point (**x**, **y**) between the points (**x0**, **y0**) and (**x1**, **y1**). String parameters to the **label** and **linemod** subroutines are terminated by null characters and must not contain new-line characters. See "plot" in topic 2.3.45 for a description of the effect of the remaining functions.

These routines appear in several separate libraries. The routines in the **libplot.a** library generate device-independent output. The **tplot** command interprets this output for a specific device.

The other versions of these routines each generate output for a specific device. You should normally redirect the output of **libprint.a** to the printer. You can save the output of **libprint.a** in a regular file and print it later. See the **tplot** command in *AIX Operating System Commands Reference* for a description of how to do this.

On an IBM Graphics Printer, the horizontal distance between points is not the same as the vertical distance between points. This means that arcs and circles are drawn as ellipses. Similarly, drawing a square (with four calls to the **line** subroutine) produces a rectangle. To adjust for this, call the **space** subroutine with appropriate scaling factors.

Files

/usr/lib/libplot.a For **tplot** filters

/usr/lib/libprint.a	For an IBM PC Graphics Printer
/usr/lib/lib300.a	For DASI 300
/usr/lib/lib300s.a	For DASI 300s
/usr/lib/lib450.a	For DASI 450
/usr/lib/lib4014.a	For Tektronix 4014
/usr/lib/librt0.a	For a vt100 terminal.
/usr/lib/libdumb.a	For a generic printer.

Related Information

In this book: "plot" in topic 2.3.45.

The **graph** and **tplot** commands in *AIX Operating System Commands Reference*.

1.2.207 popen, pclose, rpopen

Purpose
Initiates a pipe to or from a process.

Library
Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>                  #include <stdio.h>
                                    #include sys </types.h>


FILE *popen (command, type)         FILE *rpopen (command, type, site)
char *command, *type;               char *command, *type;
                                    siteno_t site;
int pclose (stream)
FILE *stream;
```

Description
The **popen** subroutine creates a pipe between the calling program and a shell command to be executed.

The **command** parameter points to a null-terminated string containing a shell command line. The **type** parameter pointers to a null-terminated string containing an I/O mode, either **r** for reading or **w** for writing.

The **popen** subroutine returns a pointer to a **FILE** structure for the stream. If the **type** parameter is **r**, you can read from the standard output of the command by reading from the file **stream**. If the **type** parameter is **w**, you can write to the standard input of the command by writing to the file **stream**.

If the Transparent Computing Facility is installed, the **rpopen** subroutine is available. The **rpopen** subroutine is similar to **popen**, but takes an additional parameter to specify the cluster site on which to run the shell command. If **site** is 0, the current site is used (which is identical to **popen**).

Use the **pclose** subroutine to close any stream you have opened with the **popen** or **rpopen** subroutine. The **pclose** subroutine waits for the associated process to terminate and then returns the exit status of the command.

Because open files are shared, a type **r** command can be used as an input filter and a type **w** as an output filter.

Warning: If the original processes and the process started with **popen** concurrently read or write a common file, neither should use buffered I/O. If they do, the results are unpredictable.

Some problems with an output filter can be prevented by taking care to flush the buffer with the **fflush** subroutine (see "fclose, fflush" in topic 1.2.77).

The **popen** and **rpopen** subroutines return a NULL pointer if files or processes cannot be created, or if the shell cannot be accessed.

The **pclose** subroutine returns -1 if **stream** is not associated with a **popen** command.

Error Conditions

The **pclose** subroutine fails if the following is true:

ECHILD The status of the child process could not be obtained.

Related Information

In this book: "fclose, fflush" in topic 1.2.77, "fopen, freopen, fdopen" in topic 1.2.82, "pipe" in topic 1.2.204, " stdio" in topic 1.2.283, "system" in topic 1.2.298, and "wait, waitpid" in topic 1.2.325.

1.2.208 printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf

Purpose
Prints formatted output.

Library
Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>

int printf ( fmt [, val, ...int NLprintf ( fmt [, val, ... ])
char *fmt;                   char *fmt;

int fprintf (stream, fmt [, int,NLfprintf (stream, fmt [, val,
FILE *stream;                FILE *stream;
char *fmt;                   char *fmt;

int sprintf (s, fmt [, val, int NLsprintf (s, fmt [, val, ...
char *s, *fmt;               char *s, *fmt;
int wsprintf (scs, fmt [,val,...])
wchar_t *wcs;
char *fmt;
```

Description
The **printf** subroutine converts, formats, and writes its **val** parameters, under control of the **fmt** parameter, to the standard output stream **stdout**.

The **fprintf** subroutine converts, formats, and writes its **val** parameters, under control of the **fmt** parameter, to the output stream specified by its **stream** parameter.

The **sprintf** subroutine converts, formats, and stores its **val** parameters, under control of the **fmt** parameter, into consecutive bytes starting at the address specified by the **s** parameter. The **sprintf** subroutine places a **'\0'** (null character) at the end. It is your responsibility to ensure that enough storage space is available to contain the formatted string.

The **wsprintf** subroutine is equivalent to **sprintf** except that the argument **wcs** specifies an array of wide characters into which the generated output is written. A NULL wide character is appended at the end of the array, but it is not counted as part of the returned sum. If copying takes place between objects that overlap, the behavior is undefined.

The **NLprintf**, **NLfprintf**, and **NLsprintf** subroutines are provided for backward compatibility and behave exactly like the **printf**, **fprintf**, and **sprintf** subroutines respectively.

The **fmt** parameter is a character string that contains two types of objects:

Plain characters, which are copied to the output stream

Conversion specifications, each of which causes zero or more items t be fetched from the **val** parameter list.

If there are not enough items for the **fmt** in the **val** parameter list, then

the results are unpredictable. If more **val**s remain after the entire **fmt** has been processed, they are ignored.

Conversions can be applied to the **n**th argument in the argument list, rather than to the next unused argument. In this case, the conversion character **%** is replaced by the sequence **%digit$**, where **digit** is a decimal integer **n** in the range of [1, {**NL_ARGMAX**}], giving the position of the argument in the argument list. With this feature, format strings can be defined to assure that arguments are selected in an order appropriate for the specified language.

Each conversion specification in the **fmt** parameter has the following syntax:

A **%** (percent) sign or the character sequence **%digit$**, which introduces the conversion specification.

Zero or more **options**, which modify the meaning of the conversion specification. The **option** characters and their meanings are:

- The result of the conversion is left-justified within the field.

+ The result of a signed conversion always begins with a sign (**+** or **-**).

blank If the first character of a signed conversion is not a sign, a blank is prefixed to the result. If both the **blank** and **+** options appear, then the **blank** option is ignored.

This option specifies that the value is to be converted to an alternate form. For **c**, **d**, **s**, and **u** conversions, the option has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a **0**. For **x** and **X** conversions, a nonzero result has **0x** or **0X** prefixed to it. For **e, E, f, g,** and **G** conversions, the result always contains a decimal point, even if no digits follow the decimal point. For **g** and **G** conversions, trailing zeros are not removed from the result.

B This option affects conversions using the **s** or **S** conversion characters of the **NLprintf**, **NLfprintf**, and **NLsprintf** subroutines only. The **B** flag specifies that **field width** and **precision** are given in bytes rather than in code points.

N This option affects the **s** and **S** conversion characters of the **NLprintf**, **NLfprintf**, and **NLsprintf** subroutines only. The **N** flag specifies that each international character support code point in the converted string converts into a printable ASCII escape sequence that uniquely identifies the code point.

An optional decimal digit string that specifies the minimum **field width**. If the converted value has fewer characters than the field width, the field is padded on the left to the length specified by the field width. If the left-adjustment option is specified, the field is padded on the right. For the **NLprintf**, **NLfprintf**, and **NLsprintf** subroutines, field width is measured in code points rather than bytes, unless the **B** flag is specified.

An optional **precision**. The precision is a **.** (period) followed by a

decimal digit string. If no **precision** is given, it is treated as 0. The **precision** specifies:

- The minimum number of digits to appear for the **d**, **i**, **u**, **o**, **x**, or **X** conversions
- The number of digits to appear after the decimal point for the **e** and **f** conversions
- The maximum number of significant digits for the **g** conversion
- The maximum number of characters to be printed from a string in the **s** conversion.

An optional **l** (the letter "ell") specifying that a following **d**, **i**, **u**, **o**, **x**, or **X** conversion character applies to a **long** integer **val**. With **printf**, an optional **L** specifying that a following **d**, **i**, **u**, **o**, **x**, or **X** conversion character applies to a **long double val**.

A character that indicates the type of conversion to be applied

% Performs no conversion. Prints a **%**.

Note: If the character after the **%** or **%digit$** sequence is not a valid conversion character, the results of the conversion are undefined.

d, i Accepts an integer **val** and converts it to signed decimal notation. The **precision** specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default **precision** is **1**. The result of converting a 0 value with a **precision** of 0 is a null string. Specifying a field width with a zero as a leading character causes the field width value to be padded with leading zeros.

Note: **i** applies to **printf** only.

u Accepts an integer **value** and converts it to unsigned decimal notation. The **precision** specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default **precision** is **1**. The result of converting a 0 value with a **precision** of 0 is a null string. Specifying a field width with a zero as a leading character causes the field width value to be padded with leading zeros.

o Accepts an integer **val** and converts it to octal notation. The **precision** specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default **precision** is **1**. The result of converting a 0 value with a **precision** of 0 is a null string. Specifying a field width with a zero as a leading character causes the field width value to be padded with leading zeros.

An octal value for field width is not implied.

x, X Accepts an integer **val** and converts it to hexadecimal notation. The letters **abcdef** are used for the **x** conversion and the letters **ABCDEF** are used for the **X** conversion. The **precision** specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded

with leading zeros. The default **precision** is **1**. The result of converting a 0 value with a **precision** of 0 is a null string. Specifying a field width with a zero as a leading character causes the field width value to be padded with leading zeros.

f Accepts a **float** or **double val** and converts it to decimal notation in the format [**-**]**ddd.ddd**. The number of digits after the decimal point is equal to the **precision** specification. If no **precision** is specified, then six digits are output. If the **precision** is 0, then no decimal point appears.

e, E Accepts a **float** or **double val** and converts it to the exponential form [**-**]**d.ddde±dd**. There is one digit before the decimal point and the number of digits after the decimal point is equal to the **precision** specification. If no **precision** is specified, then six digits are output. If the **precision** is 0, then no decimal point appears. The **E** conversion character produces a number with **E** instead of **e** before the exponent. The exponent always contains at least two digits.

g, G Accepts a **float** or **double val** and converts it in the style of the **e**, **E** or **f** conversion characters, with the **precision** specifying the number of significant digits. Trailing zeros are removed from the result. A decimal point appears only if it is followed by a digit. The style used depends on the value converted. Style **e** (**E**, if **G** is the flag used) results only if the exponent resulting from the conversion is less than -4, or if it is greater than or equal to the **precision**.

c Accepts and prints the character **val**.

C Prints one **NLchar**. Applies to **NLprintf** only.

s Accepts a **val** is as a string (character pointer) and characters from the string are printed until a **'\0'** (null character) is encountered or the number of characters indicated by the **precision** is reached. If no **precision** is specified, all characters up to the first null character are printed. If the string pointer **val** has a value of 0 or NULL, the results are undefined.

S The corresponding **NLprintf**, **NLfprintf**, or **NLsprintf val** is taken to be a pointer to a string of the type **NLchar**. Characters from the string are printed until a **\0** (null) character is encountered or the number of characters indicated by **precision** is reached. If no **precision** is specified, all characters up to the first null character are printed. If the string pointer **val** has a value of 0 or NULL, the results are undefined.

n Accepts a pointer to an integer. The number of characters successfully written so far to the stream or buffer is stored in the integer whose address is given as the argument. Applies to **printf** only.

p Accepts an integer *val* and converts it to hexidecimal notation. The *precision* specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default *precision* is 1. The result of converting a 0 value with a *precision* of 0 is a null string. Specifying a field width with a zero as a leading

character causes the field with value to be padded with zeros. This type of conversion applies to **NLprintf** only.

%wc The **wchar_t** type argument is converted to an array of characters and the resulting multibyte characters are written. If the resulting multibyte characters contain fewer bytes than the specified field width, the field is padded with blank characters. Precision is ignored, even if specified. This conversion is the same as that performed by the **wctomb** subroutine.

%ws The argument is a pointer to a **wchar_t** type array. Wide characters from the array are converted to multibyte characters and the resulting multibyte characters are written up to (but no including) a terminating NULL wide character. If both the precision and the # are specified, the number of wide characters written cannot exceed the number of bytes specified by the precision; if only precision is specified, the number of characters written is equal to the precision number. If the precision is not specified or it is greater than the size of the array, the array contains a NULL character. If the resulting multibyte characters have fewer bytes than the specified field width, the field will be padded with blanks. This conversion is the same as that performed by the **wcstombs** subroutine.

A **field width** or **precision** may be indicated by an ***** (asterisk) instead of a digit string. In this case, an integer **val** parameter supplies the field width or precision. The **val** parameter that is converted for output is not fetched until the conversion letter is reached, so the parameters specifying field width or **precision** must appear **before** the value (if any) to be converted.

If the result of a conversion is wider than the **field width**, then the field is expanded to contain the converted result. No truncation occurs. However, a small **precision** may cause truncation on the right.

The **e**, **E**, **f** and **g** formats represent the special floating-point values as follows:

Quiet NaN	**+QNaN** or **-QNaN**
Signalling NaN	**+SNaN** or **-SNaN**
±&infinity.	**+INF** or **-INF**
±0	**+0** or **-0**

The representation of the plus sign depends on whether the **+** or **blank** formatting option is specified.

Return Value

Upon successful completion, each of these subroutines except **wsprintf** returns the number of display characters in the output string rather than the number of bytes in the string. (The **NLprintf**, **NLfprintf** and **NLsprintf** subroutines use strings that may contain 2-byte **NLchar**s.) The value returned by **sprintf** and **NLsprintf** does not include the final **'\0'** character. If an output error occurs, a negative value is returned.

The **wsprintf** subroutine returns the number of wide characters written into the array, not counting the terminating NULL wide character.

Example

To print the language-dependent date and time format, the following statement could be used:

```
printf (format, weekday, month, day, hour, min);
```

For American usage, **format** could be a pointer to the string:

```
%1$s, %2$s %3$d, %4$d:%5$.2d\n
```

producing the message:

```
Sunday, July 3, 10:02
```

whereas for German usage, **format** could be a pointer to the string:

```
%1Ss, %3$d. %2$s, %4$d:%5$.2d\n
```

producing the message:

```
Sonntag, 3. Juli, 10:02
```

Related Information

In this book: "conv" in topic 1.2.50, "vprintf, vfprintf, vsprintf, NLvprintf, NLvfprintf, NLvsprintf" in topic 1.2.324, "limits.h" in topic 2.4.11, "ecvt, fcvt, gcvt" in topic 1.2.67, "putc, putchar, fputc, putw, putwc, putwchar, fputwc" in topic 1.2.213, "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf, wsscanf" in topic 1.2.241, and " stdio" in topic 1.2.283.

Examples of using **printf** in *AIX C Language Guide* and *AIX C Language Reference*.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.209 probe

Purpose
Validates the status of a TCF cluster site.

Syntax

```
#include <sys/types.h>

int probe(sitenum)
siteno_t sitenum;
```

Description
The **probe** system call tells the system to check a site to determine its status. To do this, the system sends an **are you there** message to the site. If the site does not respond, system status information is updated to indicate that the site is no longer in the TCF cluster. If the site responds, but was previously thought to be down, the distributed topology change algorithm is run to bring the site into the TCF cluster.

sitenum is the number of the site being probed.

Return Value
Upon successful completion, a value of 0 is returned to the calling process. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **probe** system call fails if one of the following is true:

EBADST **sitenum** is an invalid site number.

ESITEDN1 The requested site is down.

Related Information

In this book: "getsites" in topic 1.2.119.

1.2.210 profil

Purpose
Starts and stops execution profiling.

Syntax

```
#include <mon.h>
#include <sys/param.h>

void profil (shortbuff, bufsiz, offset, scale)
-- or --
void profil (profbuff, -1, 0, 0)

short *shortbuff;
struct prof *profbuff;
unsigned int bufsiz, offset, scale;
```

Description
The **profil** system call arranges to record a histogram of periodically sampled values of the calling process's program counter.

If the **bufsiz** parameter has any value but -1, then the parameters to **profil** are interpreted as shown in the first syntax definition. The **shortbuff** parameter points to an area of memory, and its length (in bytes) is given by the **bufsiz** parameter.

After this call, the user's program counter (pc) is examined **CLK_TCK** times a second. **CLK_TCK** is a macro defined in **<time.h>**. The value of the **offset** parameter is subtracted from the pc, and the result is multiplied by the value of the **scale** parameter. If the resulting number is less than **bufsiz** ÷ **sizeof(short)**, then the corresponding **short** inside **shortbuff** is incremented.

The least significant 16 bits of the **scale** parameter are interpreted as an unsigned, fixed-point fraction with a binary point at the left. The most significant 16 bits of **scale** are ignored. For example:

Octal	**Hex**	**Meaning**
0177777	0xFFFF	Maps approximately each pair of bytes in the instruction space to a unique **short** in **shortbuff**.
077777	0x7FFF	Maps approximately every four bytes to a **short** in **shortbuff**.
01	0x0001	Maps all instructions to the first **short** in **shortbuff**, producing a noninterrupting core clock.
0	0x0000	Turns profiling off.

Mapping each byte of the instruction space to an individual **short** in **shortbuff** is not possible.

If the second parameter (**bufsize**) has the value -1, then the parameters to **profil** are interpreted as shown in the second syntax definition. In this case, the **offset** and **scale** parameters are ignored, and **profbuff** points to an array of **prof** structures. The **prof** structure is defined in the **mon.h** header file, and it contains the following members:

```
daddr_t p_low;
daddr_t p_high;
unsigned short *p_buff;
```

```
int  p_bufsize;
int  p_scale;
```

If the **p_scale** member has the value -1, then a value for it is computed based on **p_low**, **p_high**, and **p_bufsize**; otherwise **p_scale** is interpreted like the **scale** argument in the first synopsis. The **p_high** members in successive structures must be in ascending sequence. The array of structures is terminated with a structure containing a **p_high** member set to 0.

Profiling is turned off:

- If the value of the **scale** parameter is 0.
- When an **exec** system call is executed
- If updating the buffer pointed to by the **shortbuff** or **profbuff** parameter would cause a memory fault.

Profiling is rendered ineffective by giving a value of **0** for the **bufsiz** parameter.

Profiling remains on in both the child process and the parent process after a **fork** system call.

Note: The profiling rate is machine dependent. A program should use the defined constant IHZ instead of assuming a specific value.

Related Information
In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, " fork, vfork" in topic 1.2.83, and "monitor, monstartup, moncontrol" in topic 1.2.171.

The **cc** and **prof** commands in *AIX Operating System Commands Reference*.

1.2.211 programmers workbench library

Purpose
Provides subroutines for compatibility with existing programs.

Library
Programmers Workbench Library (**libPW.a**)

Description
The **libpw** subroutines are provided only for compatibility with existing programs. Their use in new programs is not recommended.

any (c, s)
Determines whether the string **s** contains the character **c**.

anystr (s1, s2)
Determines the offset in string **s1** of the first character that also occurs in string **s2**.

balbrk (s, open, close, end)
Determines the offset in string **s** of the first character in the string **end** that occurs outside of a balanced string as defined by **open** and **close**.

cat (dest, source1,..., 0)
Concatenates the **source** strings and copy them to **dest**.

clean_up ( )
Defaults the cleanup routine.

curdir (s)
Puts the full path name of the current directory in the string **s**.

dname (p)
Determines which directory contains the file **p**.

fatal (msg)
General purpose error handler.

fdfopen (fd, mode)
Same as the **stdio fdopen** subroutine.

giveup (dump)
Forces a core dump.

imatch (pref, s)
Determines if the string **pref** is an initial substring of the string **s**.

index (s1, s2)
Determines the offset of the first occurrence in string **s1** of string **s2**.

lockit (lockfile, count, pid)
Creates a lock file.

move (s1, s2, n)
Copies the first **n** characters of string **s1** to string **s2**.

patoi (s)
Converts string **s** to **int**.

patol (s)
Converts string **s** to **long**.

rename (oldname, newname)
Renames the file **oldname** to **newname**.

Note: The **rename** subroutine is no longer part of the Programmers Workbench Library (**libPW.a**) because AIX provides a **rename** system call (see "rename" in topic 1.2.233) that performs the same function.

repeat (dest, s, n)
Sets **dest** to the string **s** repeated **n** times.

repl (s, old, new)
Replaces each occurrence of the character **old** in string **s** with the character **new**.

satoi (s, ip)
Converts string **s** to int and save it in ***ip**.

setsig ( )
Causes signals to be caught by **setsig1**.

setsig1 (sig)
General purpose signal handling routine.

sname (s)
Gets a pointer to the simple name of full path name **s**.

strend (s)
Finds the end of the string **s**.

substr (s, dest, origin, len)
Places a substring of string **s** in **dest** using the offset **origin** and the length **len**.

trnslat (s, old, new, dest)
Copies string **s** into **dest** and replace any character in **old** with the corresponding characters in **new**.

unlockit (lockfile, pid)
Deletes the lock file.

userdir (uid)
Gets the user's login directory.

userexit (code)
Defaults user exit routine.

username (uid)
Gets the user's login name.

verify (s1, s2)
Determines the offset in string **s1** of the first character that is not also in string **s2**.

xalloc (asize)
Allocates memory.

xcreat (name, mode)
Creates a file.

xfree (aptr)
Frees memory.

xfreeall ( )
Frees all memory.

xlink (f1, f2)
Links files.

xmsg (file, func)
Calls the routine **fatal** with an appropriate error message.

xopen (name, mode)
Opens a file.

xpipe (t)
Creates a pipe.

xunlink (f)
Removes a directory entry.

xwrite (fd, buffer, n)
Writes **n** bytes to the file associated with **fd** from **buffer**.

zero (p, n)
Zeros **n** bytes starting at address **p**.

zeropad (s)
Replaces the initial blanks with the character **'0'** in string **s**.

In addition, NLS equivalents exist for the following routines:

NCany	**NCmove**	**NCtrnslat**
NCanystr	**NCpatoi**	**NCverify**
NCbalbrk	**NCrepeat**	**NCzero**
NCcat	**NCrepl**	**NCzeropad**
NCimatch	**NCsatoi**	
NCindex	**NCstrend**	
	NCsubstr	

These routines perform identical functions as the non-NLS routines except that they deal with **NLchars** rather than ordinary chars.

Related Information
In this book: "logname" in topic 1.2.159, "NLchar" in topic 1.2.188, and "regcmp, regex" in topic 1.2.228.

1.2.212 ptrace

Purpose
Traces the execution of another process.

Syntax

```
#include <sys/ptrace.h>

int ptrace (request, pid, addr, data, buff)
int request;
pid_t pid;
int *addr, data;
char *buff;
```

Description
The **ptrace** system call allows one process to control the execution of another process. **ptrace** is primarily used by utility programs to implement breakpoint debugging. **ptrace** does not support multiprocessor debugging. The **dbx** command described in *AIX Operating System Commands Reference* is one such debugging utility.

The traced process behaves normally until it encounters a signal, at which time it enters a stopped state and its debugging process is notified with the **wait** system call. When the traced process is in the stopped state, its debugging process can examine and modify its memory image using the **ptrace** system call. Also, the debugging process can cause the traced process to either terminate or continue, with the possibility of ignoring the signal that caused it to stop.

If the Transparent Computing Facility is installed, the traced process must be a process executing on the local site. This process is also forbidden to migrate to another site.

The **request** parameter determines the action to be taken by the **ptrace** system call and is one of the following:

PT_TRACE_ME(0)
This request must be issued by a child process that is to be traced by its parent. This request sets the child process's trace flag that causes the child to be left in a stopped state upon receipt of a signal, rather than the state specified by the **func** parameter of the **signal** system call. The **pid**, **addr**, and **data** parameters are ignored, and a return value is not defined for this request. Do not issue this request if the parent does not expect to trace the child.

Note: The remainder of the requests can only be used by the debugging process. For each request, the **pid** parameter is the process ID of the process to be traced. The traced process must be in a stopped state before these requests are made.

PT_READ_I(1), PT_READ_D(2)
These requests return the **int** value in the traced process's address space at the location pointed to by the **addr** parameter. **PT_READ_I** specifies the text area while **PT_READ_D** specifies the data area. The **data** parameter is ignored. These requests fail if the value of the **addr** parameter is not in the address space of the traced process, in which case a value of -1 is returned, and the debugging process's **errno** is set to EIO.

PT_READ_U(3)
This request returns the **int** value from the traced process's user area of the system's address space that is located at the offset given by the **addr** parameter. (For information about the user area, see the **sys/user.h** header file.) The value of the **addr** parameter must be in the range 0 to **ctob(USIZE)**, and it is rounded down to the next **int** (word) boundary. (**ctob** and **USIZE** are defined by including the **sys/param.h** header file.) The **data** parameter is ignored. This request fails if the **addr** parameter is outside the user area, in which case a value of -1 is returned to the debugging process, whose **errno** is then set to EIO.

PT_WRITE_I(4), PT_WRITE_D(5)
These requests write the value of the **data** parameter into the address space of the traced process at the **int** location pointed to by the **addr** parameter. Request **PT_WRITE_I** writes into the text area and request **PT_WRITE_D** writes into the data area. Upon successful completion, the value written into the address space of the traced process is returned to the debugging process. These requests fail if the **addr** parameter points to a location in a pure procedure space (read-only) and a copy cannot be made. They also fail if the value of **addr** is out of range. Upon failure, a value of -1 is returned to the debugging process, and the debugging process's **errno** is set to EIO.

PT_WRITE_U(6)
This request writes the value of the **data** parameter into the child process's user area of the system's address space at the **int** location specified by the **addr** parameter. The value of the **addr** parameter is rounded down to the next **int** (word) boundary. The following values for **addr** are defined in the **sys/ptrace.h** header file, and they identify the only entries that can be modified.

PT_CONTINUE(7)
This request causes the traced process to resume execution. If the **data** parameter is **0**, all pending signals, including the one that caused the traced process to stop, are canceled before the traced process resumes execution. If the **data** parameter is a valid signal number, the traced process resumes execution as if it had received that signal. Any other pending signals are canceled. If the **addr** parameter is equal to **1** for this request, then execution continues from where it left off. Otherwise, execution continues from the address specified in **addr**. Upon successful completion, the value of the **data** parameter is returned to the debugging process. This request fails if the **data** parameter is not 0 or a valid signal number, in which case a value of -1 is returned to the debugging process and the debugging process's **errno** is set to EIO.

PT_KILL(8)
This request causes the traced process to terminate the same way it would if it had received the **SIGKILL** signal.

PT_STEP(9)
This request puts the traced process into trace mode and then executes the same steps as listed for request **PT_CONTINUE**. Trace mode causes the traced process to stop upon completion of one machine instruction, which allows single stepping of the traced process. The signal number from the stop is **SIGTRAP**.

PT_READ_GPR(11)
This request returns the contents of one of the general purpose registers of the traced process. The **addr** parameter specifies which of the registers is to be returned. The **data** and **buff** parameters are ignored. The registers are specified using the macro **PT_REG**. The argument to **PT_REG** is the symbolic name found in **<sys/ptrace.h>**. The request fails if the **addr** parameter does not specify a legitimate register. In this case, **ptrace** returns the value -1 and sets the debugging process's **errno** to EIO.

PT_READ_FPR(12)
This request stores the value of a floating-point register into the location pointed to by the **addr** parameter. The **data** parameter specifies which floating-point register, and its value should be the offset into the trace process's saved floating point state. The format of this structure is **struct fp87save**, found in **<sys/ptrace.h>**. Floating point registers are ten bytes long.

PT_WRITE_GPR(14)
This request stores the value of the **data** parameter in one of the traced process's general purpose registers. The **addr** parameter specifies the register to be modified. The **buff** parameter is ignored. Upon successful completion, the value of **data** is returned to the debugging process. The registers are specified in the same manner as used for **PT_READ_GPR**. This request fails if the **addr** parameter does not specify a legitimate register. In this case, **ptrace** returns the value -1 and sets the debugging process's **errno** to EIO.

PT_WRITE_FPR(15)
This request sets the floating-point register specified by the **data** parameter to the value pointed to by the **addr** parameter. The **data** parameter is specified in the same manner as with **PT_READ_FPR**. Floating point registers are ten bytes long.

PT_READ_BLOCK(17)
This request reads a block of data from the traced process's address space. The **addr** parameter points to the block of data in the traced process's address space and the **data** parameter gives its length in bytes. The value of the **data** parameter must not be greater than 1024. The **buff** parameter points to the location in the debugging process's address space into which the data is to be copied. Upon successful completion, **ptrace** returns the value of the **data** parameter. If an error occurs, **ptrace** returns -1 and sets the debugging process's **errno** to indicate the error.

This request fails when one or more of the following are true:

EINVAL The **data** parameter is less than 1 or greater than 1024.

EIO The **addr** parameter is not a valid pointer into the traced process's address space.

EFAULT The **buff** parameter does not point to a writable location in the debugging process's address space.

PT_WRITE_BLOCK(19)
This request writes a block of data into the traced process's address space. The **addr** parameter points to the location in the traced process's address space to be written into. The **data** parameter gives

the length of the block in bytes, and it must not be greater than 1024. The **buff** parameter points to the data in the debugging process's address space to be copied. Upon successful completion, the value of **data** is returned to the debugging process. If an error occurs, **ptrace** returns -1 and sets the debugging process's **errno** to indicate the error. This request fails when one or more of the following are true:

EINVAL The **data** parameter is less than 1 or greater than 1024.

EIO The **addr** parameter is not a valid pointer into the traced process's address space.

EFAULT The **buff** parameter does not point to a readable location in the debugging process's address space.

If the process has never accessed the vector hardware, a read will return a buffer with all locations filled with zeros. The following describes in detail the function of each of the 11 requests that have been added. Specific values for the macros described can be found in Appendix A, **ptrace.h**.

PT_READ_VSEG_SIZ
Read the vector segment size (20)

This request returns the vector section size; *Pid* specifies the specific processes that the action should be performed on. The *addr*, *data* and *buff* parameters are ignored. If the cpu has no vector facility, then the request returns the value -1 and the debugging process's **errno** will be set to **EINVAL**. If there are no errors, a zero will be returned.

PT_READ_VSR
Read the vector status register (21)

This request reads the value of the vector status register and stores it into the location pointed to by the addr parameter. The vector status register is 8 bytes long. *Pid* specifies the specific process that the action should be performed on. The *data* and *buff* parameters are ignored.

If an error occurs, **ptrace** returns -1 and sets the debugging process's errno to indicate the error. The request fails when one of the following are true:

ENODEV The cpu has no vector facility.

EFAULT A detectable memory access error has occurred during operation.

If there are no errors, a zero will be returned.

PT_WRITE_VSR
Write to the vector status register (22)

This request stores the value pointed to by the *addr* parameter into the vector status register. The vector status register is 8 bytes long. *Pid* specifies the specific process that the action should be performed on. The *data* and *buff* parameters are ignored.

If an error occurs, **ptrace** returns -1 and sets the debugging process's errno to indicate the error. The request fails when one of the following are true:

EACCES An attempt was made to alter bits 0-15 or bits 56-63 of the status register, which are protected.

ENXIO The values given for the vector count and/or the vector interruption index are greater than the section size.

PT_READ_VMR
Read the vector mask register (23)

This request reads the vector mask register and stores its value into the location pointed to by the addr parameter. The vector mask register is vector segment size number of bits long. *Pid* specifies the specific process that the action should be performed on. The *data* and *buff* parameters are ignored.

PT_WRITE_VMR
Write to the vector mask register (24)

This request stores the value pointed to by the addr parameter into the vector mask register. The vector mask register is vector segment size number of bits long. *Pid* specifies the specific process that the action should be performed on. The *data* and *buff* parameters are ignored.

PT_READ_VACR
Read the vector activity register (25)

This request reads the vector activity register and stores its value into the location pointed to by the addr parameter. The vector activity register is 8 bytes long. *Pid* specifies the specific process that the action should be performed on. The *data* and *buff* parameters are ignored.

PT_WRITE_VACR
Write the vector activity register (26)

This request stores the value pointed to by the addr parameter into the vector activity register. The vector activity register is 8 bytes long. *Pid* specifies the specific process that the action should be performed on. The *data* and *buff* parameters are ignored.

EACCES An attempt was made to alter bits 0-8 of the vector activity register, which are protected.

PT_READ_VFR
Read a 32-bit vector register (27)

This request reads the values of the 32-bit vector register specified by the addr parameter and stores its values into the location specified by the buff parameter. The length of the data is equal to 4 times the vector segment size in bytes. Register numbers are specified in the header file ptrace.h. Either an odd or an even number can be specified in this request. Pid specifies the specific process that the action should be performed on. The data parameter is ignored.

If an error occurs, ptrace returns -1 and sets the debugging process's errno to indicate the error. The request fails when one of the following are true:

ENODEV The cpu has no vector facility

ENXIO The register number is out of the range of numbers PT_R_V0 to PT_R_V15.

EFAULT A detectable memory access error has occurred during operation, (e.g, the memory area pointed to by the buff parameter is less than 4 times the vector segment size in bytes, and the copy of the vector would write outside of user space.

If there are no errors, a zero will be returned.

PT_WRITE_VFR
Write into the 32-bit vector register (28)

This request stores the values pointed to by the buff parameter into the 32-bit vector register specified by the addr parameter. The length of the data is equal to 4 times the vector segment size in bytes. Register numbers are specified in the header file ptrace.h. Either an odd or an even number can be specified in this request. Pid specifies the specific process that the action should be performed on. The data parameter is ignored.

PT_READ_VFR
Read the double (64-bit) vector register (29)

This request reads the even-odd pair of vector registers specified by the addr parameter and stores its values into the location specified by the buff parameter. The length of the data is equal to 8 times the vector segment size in bytes. Register numbers are specified in the header file ptrace.h. Only an even number can be specified for the vector register's pair in the request. Pid specifies the specific process that the action should be performed on. The data parameter is ignored.

If an error occurs, ptrace returns -1 and sets the debugging process's errno to indicate the error. The request fails when one of the following are true.

ENODEV The cpu has no vector facility.

ENXIO The register number is out of the range of even numbers PT_R_V0 to PT_R_V14.

EFAULT A detectable memory access error has occurred during operation, (e.g, the memory area pointed to by the buff parameter is less than 8 times the vector segment size in bytes, and the copy of the vector would write outside of user space.

If there are no errors, a zero will be returned.

PT_WRITE_VDR
Write into the double (64-bit) vector register (30)

This request stores the values pointed to by the buff parameter into the even-odd pair of registers specified by the addr parameter. The length of the data is equal 8 times the vector segment size in bytes. Register numbers are specified in the header file ptrace.h. Only an even number can be specified for the vector register's pair in this request. *Pid* specifies the specific process that the action should be performed on. The *data* parameter is ignored.

OTHER ERRORS

In general, the **ptrace** system call will fail if one or more of the following are true:

EIO The request parameter is not one of the values listed.

ESRCH The *pid* parameter identifies a child process that does not exist or has not executed a **ptrace** system call with request 0.

As a security measure, the **ptrace** system call inhibits the set-user-ID facility on subsequent **exec** system calls.

If a traced process initiates an **exec** system call, it stops before executing the first instruction of the new image and shows the signal **SIGTRAP**.

Error Conditions

In general, the **ptrace** system call fails if one or more of the following are true:

EIO The **request** parameter is not one of the values listed.

ESRCH The **pid** parameter identifies a child process that does not exist on the local system or that has not executed a **ptrace** system call with request 0.

EAGAIN The traced process was not in a stopped state before a request is made.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, "sigaction, sigvec, signal" in topic 1.2.263, and "wait, waitpid" in topic 1.2.325.

The **dbx** command in *AIX Operating System Commands Reference*.

1.2.213 putc, putchar, fputc, putw, putwc, putwchar, fputwc

Purpose
Writes a character or a word to a stream.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <stdio.h>

int putc(c, stream)
char c;
FILE *stream;

int putchar(c)
char c;

wchar_t putwc (c, stream)
wchar_t c;
FILE *stream;

wchar_t putwchar (c)
wchar_t c;

int fputc(c, stream)
char c;
FILE *stream;

int putw(w, stream)
int w;
FILE *stream;

wchar_t fputwc (c, stream)
wchar_t c;
FILE *stream;
```

Description
The **putc** macro writes the character **c** to the output specified by the **stream** parameter. The character is written at the position at which the file pointer is currently pointing, if defined.

The **putchar** macro is the same as the **putc** macro except that **putchar** writes to the standard output.

The **fputc** subroutine works the same as **putc**, but **fputc** is a true subroutine rather than a macro. It runs more slowly than **putc**, but takes less space per invocation.

Because **putc** is implemented as a macro, it treats incorrectly a **stream** parameter with side effects, such as **putc(c, *f++)**. For such cases, use **fputc** instead. Also, use **fputc** whenever you need to pass a pointer to this subroutine as a parameter to another subroutine.

The **putw** subroutine writes the word (**int**) specified by the **w** parameter to the output specified by the **stream** parameter. The word is written at the position at which the file pointer, if defined, is pointing. The size of a word is the size of an integer and varies from machine to machine. The **putw** subroutine does not assume or cause special alignment of the data in the file.

Because of possible differences in word length and byte ordering, files written using the **putw** subroutine are machine-dependent, and may not be readable using the **getw** subroutine on a different processor.

With the exception of **stderr**, output streams are, by default, buffered if they refer to files, or line-buffered if they refer to terminals. The standard error output stream, **stderr**, is unbuffered by default, but using the **freopen** subroutine causes it to become buffered or line-buffered. Use

the **setbuf** subroutine to change the stream's buffering strategy.

When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as it is written. When an output stream is buffered, many characters are saved and written as a block. When an output stream is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested).

The **putwchar** subroutine returns the character written. If a write error occurs, the error indicator for the stream is set and **putwchar** returns WEOF.

The **fputwc** subroutine writes the character specified by **c** to the output stream pointed to by **stream**, as a multibyte character at the position indicated by the associated file position indicator for the stream (if defined), and it advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

Return Value
Upon successful completion, each of these functions (with the exception of **putw**) returns the value it has written. **putw** returns **ferror** (stream). If these functions fail, they return the constant EOF. They fail if the **stream** is not open for writing, or if the output file size cannot be increased. Because EOF is a valid integer, you should use the **ferror** subroutine to detect **putw** errors.

The **putwc** subroutine returns the argument path to it. If a write error occurs, the error indicator for the stream is set and **putwc** returns WEOF.

The **putwchar** subroutine is equivalent to **putwc** with the second argument **stdout**.

The **fputwc** subroutine returns the wide character written. If a write error occurs, the error indicator for the stream is set and **fputwc** returns WEOF.

Error Conditions
The **putc**, **putchar**, **fputc**, and **putw** subroutines fail if one or more of the following are true:

EAGAIN The **O_NONBLOCK** flag is set for the file descriptor underlying **stream** and the process is delayed in the write operation.

EBADF The file descriptor underlying **stream** is not a valid file descriptor open for writing.

Note: If a wide character routine fails and **errno** is not set, this indicates that the translation from wide code to file code has failed.

EFBIG An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

EINTR The write operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfers for this file.

EIO The implementation supports job control, the process is a member of a background process group attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking **SIGTTOU** and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.

ENOSPC There was no free space remaining on the device containing the file.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device.

Related Information

In this book: "fclose, fflush" in topic 1.2.77, "feof, ferror, clearerr, fileno" in topic 1.2.79, "fopen, freopen, fdopen" in topic 1.2.82, "fread, fwrite" in topic 1.2.84, "getc, fgetc, getchar, getw, getwc, fgetwc, getwchar" in topic 1.2.91, "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf" in topic 1.2.208, "puts, fputs, putws, fputws" in topic 1.2.216, "setbuf, setvbuf" in topic 1.2.247, and " stdio" in topic 1.2.283.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.214 putenv

Purpose
Sets an environment variable.

Library
Standard C Library (**libc.a**)

Syntax

```
int putenv (str)
char *str;
```

Description
The **putenv** subroutine sets the value of an environment variable by altering an existing variable or by creating a new one. The **str** parameter points to a string of the form **name=value**, where **name** is the environment variable and **value** is the new value for it.

The memory space pointed to by the **str** parameter becomes part of the environment, so that altering the string effectively changes part of the environment. The space is no longer used after the value of the environment variable is changed by calling **putenv** again.

Warning: Unpredictable results can occur if a subroutine passes **putenv** a pointer to an automatic variable and then returns while the variable is still part of the environment.

Note: The **putenv** subroutine manipulates the environment pointed to by the **environ** external variable, and it can be used in conjunction with **getenv**. However, **envp**, the third parameter to **main**, is not changed. See "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71 for more information about **environ** and **envp**.

The **putenv** subroutine uses **malloc** to enlarge the environment.

After **putenv** is called, environment variables are not necessarily in alphabetical order.

Return Value
Upon successful completion, a value of 0 is returned. If **malloc** is unable to obtain sufficient space to expand the environment, then **putenv** returns a value of -1.

Error Conditions
The **putenv** subroutine fails if the following is true:

ENOMEM Insufficient memory was available.

Related Information
In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, "getenv, NLgetenv" in topic 1.2.94, "malloc, free, realloc, calloc, valloc, alloca, mallopt, mallinfo" in topic 1.2.162, and "environment" in topic 2.4.6.

1.2.215 putpwent

Purpose
Writes a password file entry.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;
```

Description
The **putpwent** subroutine writes a line on the stream specified by the **f** parameter. The stream that is written on matches the format of **/etc/passwd**.

The **p** parameter is a pointer to a **passwd** structure created by the **getpwent**, **getpwuid**, or **getpwnam** subroutines.

Return Value
Upon successful completion, **putpwent** returns a value of 0. If **putpwent** fails, a nonzero value is returned.

Related Information
In this book: "getpwent, getpwuid, getpwnam, setpwent, endpwent" in topic 1.2.114 and "passwd" in topic 2.3.44.

1.2.216 puts, fputs, putws, fputws

Purpose
Writes a string to a stream.

Library
Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>

int puts (s)                    int fputs (s, stream)
char *s;                        char *s;
                                *stream;
int putws (s)
const wchar_t *s;

int fputws (s, stream);

const wchar_t *s;
FILE *stream;
```

Description
The **puts** subroutine writes the NULL-terminated string pointed to by the **s** parameter, followed by a new-line character, to the standard output stream, **stdout**.

The **fputs** subroutine writes the NULL-terminated string pointed to by the **s** parameter to the output stream specified by the **stream** parameter. The **fputs** subroutine does not append a new-line character.

Neither subroutine writes the terminating NULL character.

The **putws** subroutine writes the character string pointed to by **s** to the stream pointed to by **stdout** as a multibyte character string and appends a new-line character to the output. The terminating NULL character is not written.

The **fputws** subroutine writes the string pointed to by **s** to the stream pointed to by **stream** as a multibyte character string. The terminating NULL character is not written.

Return Value
Upon successful completion, the **puts** and **fputs** subroutines return the number of characters written. Both subroutines return EOF on an error. This happens if the routines try to write on a file that has not been opened for writing.

The **putws** and **fputws** subroutines return -1 if a write error occurs; otherwise, they return a non-negative value.

Error Conditions
The **puts** and **fputs** subroutines fail if one or more of the following conditions are true:

EAGAIN The **O_NONBLOCK** flag is set for the file descriptor underlying **stream** and the process is delayed in the write operation.

EBADF The file descriptor underlying **stream** is not a valid file descriptor open for writing.

Note: If a wide character routine fails and **errno** is not set, this indicates that the translation from wide code to file code has failed.

EFBIG An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

EINTR The write operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfers for this file.

EIO The implementation supports job control, the process is a member of a background process group attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking **SIGTTOU** and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.

ENOSPC There was no free space remaining on the device containing the file.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device.

EPIPE An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal is also sent to the process.

ENOMEM Insufficient storage space is available.

Related Information
In this book: "feof, ferror, clearerr, fileno" in topic 1.2.79, "fopen, freopen, fdopen" in topic 1.2.82, "fread, fwrite" in topic 1.2.84, "gets, fgets, getws, fgetws" in topic 1.2.117, "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf" in topic 1.2.208, "putc, putchar, fputc, putw, putwc, putwchar, fputwc" in topic 1.2.213, and " stdio" in topic 1.2.283.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.217 qsort

Purpose
Sorts a table of data in place.

Library
Standard C Library (**libc.a**)

Syntax

```
void qsort (base, nel, size, compar)
void * base;
size_t nel, size;
int (*compar)  (void *, void *);
```

Description
The **qsort** subroutine sorts a table of data in place. It uses the "quicker-sort" algorithm.

The **base** parameter points to the element at the base of the table. The **nel** parameter is the number of elements in the table. The **compar** parameter is the name of the comparison function. (See "alphasort" in topic 1.2.15 for one such comparison function.)

The comparison function must compare its parameters and return a value as follows:

- If the first parameter is less than the second parameter, **compar** must return a value less than 0.
- If the first parameter is equal to the second parameter, **compar** must return 0.
- If the first parameter is greater than the second parameter, **compar** must return a value greater than 0.

The comparison function need not compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

Note: The order in the output of two items that compare equal is unpredictable.

The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

Related Information
In this book: "alphasort" in topic 1.2.15, "bsearch" in topic 1.2.23, "lsearch, lfind" in topic 1.2.160, and "string" in topic 1.2.288.

The **sort** command in *AIX Operating System Commands Reference*.

1.2.218 quota

Purpose
Manipulate disk quotas.

Syntax

#include <sys/quota.h>

int quota (cmd, uid, arg, addr)
int cmd, uid, arg;
char *addr;

Description
The **quota** system call manipulates disk quotas for file systems that have had quotas enabled with **setquota**. The **cmd** parameter indicates a command to be applied to the user ID, specified by the **uid** parameter. The **arg** parameter is a command-specific argument, and the **addr** parameter specifies the address of an optional, command-specific, data structure that is copied in or out of the system. interpretation of the **arg** and **address** parameters is given with each command as follows:

Q_SETDLIM Set disk quota limits and current usage for the user specified by the **uid** parameter. The **arg** parameter specifies a major/minor device indicating a particular file system. The **addr** parameter is a pointer to a **struct dqblk**, as defined in **sys/quota.h**.

Only superusers may set quota limits.

Q_GETDLIM Get disk quota limits and current usage. Parameters are the same as for **Q_SETDLIM**.

This command is unprivileged.

Q_SETDUSE Set disk usage limits for a user. Parameters are the same as for **QSETDLIM**, except that addr points to a **struct dqusage** structure.

Only superusers may set usage limits.

Q_SYNC Update the on-disk copy of quota usages. The **arg** indicates the major/minor device number of the file system to be updated. If the **arg** parameter is specified as **NODEV**, all file systems that have disk quotas will be updated. The **uid** and **addr** parameters are ignored.

This command is unprivileged.

Q_SETUID Change the calling process's quota limits to those of the user with ID **uid**. The **arg** and **addr** parameters are ignored.

Only superusers may set the quota user ID.

Q_SETWARN Alter the disk usage warning limits for the user with ID **uid**. The **arg** parameter specifies the major/minor device number of a particular file system to which this command is to be applied. The **addr** parameter is a pointer to a struct **dqwarn structure**.

Only superusers may set the usage warning limits.

Q_DOWARN Warn the user specified by the **uid** parameter about excessive disk usage. This call causes the system to check its current disk usage information and print a message on the terminal of the user, if the user is over quota. If the user is under quota, his warning count is reset to **MAX*WARN** (defined in **sys/quota.h**). If the **arg** parameter specifies **NODEV**, all file systems that have disk quotas will be checked. Otherwise, **arg** specifies a specific file system's major/minor device number.

Only superusers may send warnings.

Return Value
A 0 return value indicates that the call succeeded. A return value of -1 indicates that an error occurred, and an error code is stored in the global variable **errno**.

Error Conditions
If **quota** system call fails if one or more of the following are true:

EINVAL The system is not configured to support **quota** option.

EINVAL The **cmd** parameter is invalid.

ESRCH No disk quota is found for the user ID **uid**.

EPERM The **cmd** parameter requires privilege, and the calling process's effective user ID does not have superuser privileges.

ENODEV The **arg** parameter indicates an invalid or unmounted file system.

EFAULT The **addr** parameter points to a location outside of the process's allocated address space.

EUSERS The quota table is full.

Related Information
In this book: "getrlimit, setrlimit, vlimit" in topic 1.2.115, "getrusage, vtimes" in topic 1.2.116, "setquota" in topic 1.2.253, and "ulimit" in topic 1.2.313.

The discussion of **quotacheck** and **quotaon** in *AIX Operating Systems Commands Reference*.

1.2.219 raccept

Purpose
Accepts a replicated file system recovery queue entry.

Syntax

```
#include <sys/types.h>
#include <sys/recovery.h>

int raccept(rcmdp)
struct rcmd *rcmdp;
```

Description
The **raccept** system call returns the first recovery entry from the kernel queue to the location pointed to by the **rcmdp** pointer. If the queue is empty, the requesting process sleeps until the queue is not empty. It is only intended for use by the replicated file system recovery master process, which waits for a request, then forks the appropriate recovery software. Other use may disrupt its operation.

The **raccept** system call can only be used by the superuser.

Error Conditions
The following error codes are returned if **raccept** fails and -1 is returned:

EPERM The effective user ID of the calling process is not superuser.

EFAULT **rcmdp** is not contained in the allocated address space.

EINTR A signal was caught while the process was waiting for a request.

Related Information
In this book: "chlwm" in topic 1.2.43 and "spropin" in topic 1.2.279.

The **primrec** and **recmstr** commands in *AIX Operating System Commands Reference*.

1.2.220 raise

Purpose
Sends a signal to a running program.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <signal.h>

int raise (sig);
int sig;
```

Description
The **raise** function sends the signal **sig** to a running program. Upon completion, the return value is 0 if successful and nonzero if unsuccessful.

Example
The following example requests termination by raising condition **SIGTERM**.

```
#include <signal.h>

main ()
{

    raise (SIGTERM);     /*  Request termination  */
}
```

1.2.221 rand, srand

Purpose
Generates pseudo-random numbers.

Library
Standard C Library (**libc.a**)
Berkeley Compatibility Library (**libbsd.a**)

Syntax

```
int rand ( )                    void srand (seed);
                                unsigned int seed;
```

Description
The **rand** subroutine generates random numbers using a multiplicative congruential algorithm. The random-number generator has a period of 2(32), and it returns successive pseudo-random numbers in the range from 0 to 2(15) - 1.

The **srand** subroutine resets the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

Note: The **rand** subroutine is a very simple random-number generator. Its spectral properties, the mathematical measurement of how "random" the number sequence is, are somewhat weak. See "drand48" in topic 1.2.63 or "random, srandom, initstate, setstate" in topic 1.2.222 for a more elaborate random-number generator that has better spectral properties.

Compatibility Note

The 4.3BSD version of **rand** returns a number in the range 0 to 2(31)-1, rather than 0 to 2(15)-1, and can be used by compiling with the Berkeley Compatibility Library (**libbsd.a**). There are better random number generators, as noted above, however **rand** and **srand** are the interfaces defined for the ANSI C Library. The following functions define the semantics of **rand** and **srand**, and are included here to facilitate porting applications from different implementations:

```
static unsigned int next = 1;

int rand ( )
{
 next = next * 1103515245 + 12345;
 return ( (next/65536) % 32768);
}

void srand (seed)
int seed;
{
 next = seed;
}
```

1.2.222 random, srandom, initstate, setstate

Purpose
Generates pseudo-random numbers and changes number generators.

Library
Standard C Library (**libc.a**)

Syntax

```
long random ( )

srandom (seed);
int seed;
char *setstate (state)
char *state;

char *initstate (seed, state, byte
unsigned int seed;
char *state;
int bytes;
```

Description
The **random** subroutine generates random numbers using a nonlinear additive feedback random-number generator. This generator uses a default table size of 31 long integers to return successive pseudo-random numbers in the range from 0 to 2(31)-1. The period of this random-number generator is very large, approximately 16¦ (2(31)-1).

The **srandom** subroutine resets the random-number generator to a random starting point when used after the **setstate** subroutine. Like the **rand** subroutine, the **random** generator is initially seeded with a value of 1.

The **srandom** subroutine, unlike the **srand** subroutine, does not return the old seed because the amount of state information used is more than a single word. Two other subroutines, **initstate** and **setstate**, handle restarting and changing random-number generators.

The **initstate** subroutine allows a state array, passed in by the **state** parameter, to be initialized for future use. The size of the state array (in bytes) is contained in the **bytes** parameter, which is used by the **initstate** subroutine to decide how complex the random-number generator should be. When more bytes of state information are used, the numbers are more random. Values for the amount of state information are: 8, 32, 64, 128, and 256 bytes. Amounts less than 8 bytes generate an error, while other amounts are rounded down to the nearest known value. The **seed** parameter specifies a starting point for the random-number sequence and provides for restarting at the same point. The **initstate** subroutine returns a pointer to the previous state information array.

Once a state has been initialized, the **setstate** subroutine allows rapid switching between states. The array defined by **state** parameter is used for further random-number generation until the **initstate** subroutine is called or the **setstate** subroutine is called again. The **setstate** subroutine returns a pointer to the previous state array.

After initialization, a state array can be restarted at a different point in one of two ways:

The **initstate** subroutine can be used, with the desired seed, state array, and size of the array, or

The **setstate** subroutine, with the desired state, can be used, followed by the **srandom** subroutine with the desired seed. The advantage of using both of these subroutines is that the size of the state array

does not have to be saved once it is initialized.

With a full 256 bytes of state information, the period of the random-number generator is greater than 2(69), which should be sufficient for most purposes.

Error Conditions

If the **initstate** subroutine is called with less than 8 bytes of state information, or if the **setstate** subroutine detects that the state information has been damaged, error messages are sent to the standard output.

Related Information

In this book: "drand48" in topic 1.2.63 and "rand, srand" in topic 1.2.221.

1.2.223 rcmd, rresvport, ruserok

Purpose
Allows execution of commands on a remote host that is running the **rshd**.

Library
Internet Library (**libc.a**)

Syntax

```
int rcmd (host, port, locuser, remuser, command, errfd)
char **host;
unsigned short port;
char *locuser, *remuser, *command;
int *errfd;

int rresvport (port)
int *port;

int ruserok (host, superuser, remuser, locuser)
char *host;
int superuser;
char *remuser, *locuser;
```

Description

The **rcmd** subroutine is used to execute a command on a remote machine. The **rresvport** subroutine is used to obtain a socket with a privileged address bound to the socket. A ***privileged*** Internet port is one that falls in the range 0 to 1023. The **ruserok** subroutine is used by servers to authenticate clients requesting services with the **rcmd** subroutine.

The **rcmd** and **rresvport** subroutines can only be used by processes with an effective user ID of superuser. An authentication scheme based on remote port numbers is used to verify permissions.

The **rcmd** subroutine uses the **gethostbyname** subroutine to find the host specified by **host**. The **host** parameter is updated to point to the standard name of the host found by **gethostbyname**. If the host does not exist, the **rcmd** subroutine fails and returns -1.

The **port** parameter specifies the ***well-known*** DARPA Internet port to use for the connection, which is part of the services data base. (See the description of the **/etc/services** file in *AIX TCP/IP User's Guide* for more information.)

The **locuser** and **remuser** parameters point to user names that are valid at the local and remote host, respectively. Any valid user name can be given.

The **command** parameter points to the name of the command to be executed at the remote host.

If the connection succeeds, a socket in the Internet domain of type **SOCK_STREAM** is returned to the calling process and given to the remote command as standard input and standard output.

If **errfd** is not 0, an auxiliary channel to a control process is set up, and the **errfd** parameter points to the file descriptor for the channel. The control process provides diagnostic output from the remote command on

this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command.

If **errfd** is NULL, then the standard error of the remote command is the same as standard output, and no provision is made for sending arbitrary signals to the remote process. In this case, however, it may be possible to send out-of-band data to the remote command.

The **host** parameter of the **ruserok** subroutine contains the name of a remote host. The **ruserok** subroutine checks for this host in the **/etc/host.equiv** file. Then, if necessary, this subroutine checks a file in the user's home directory at the server called **.rhosts** for a host and remote user ID.

The **superuser** parameter indicates whether the effective user ID of the calling process is that of the superuser. A value of 0 indicates the caller is not superuser. A value of 1 indicates that this process has local superuser privileges, and the checking of the **/etc/host.equiv** file is not performed.

The **remuser** and **locuser** parameters point to user names that are valid at the local and remote host, respectively. Any valid user name can be given.

If the local domain (obtained with the **gethostname** subroutine) is the same as the remote domain, only the host name (without the domain parts) must be specified.

Return Value

The **rcmd** subroutine returns a valid socket descriptor on success. If the effective user ID of the calling process is not superuser, **rcmd** returns a value of -1.

The **rresvport** subroutine returns a valid, bound socket descriptor on success. If the **rresvport** subroutine fails, a value of -1 is returned and **errno** is set to indicate the error.

The **ruserok** subroutine returns a value of 0 if the **host** name is found in the **/etc/hosts.equiv** file, or if the **host** and **remuser** IDs are found in the **.rhosts** file. If the **host** is not found, **ruserok** returns a value of -1.

Error Conditions

The **rresvport** subroutine fails if one or more of the following are true:

EAGAIN	All network ports are in use.
EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
EMFILE	Two hundred (200) file descriptors are currently open.
ENFILE	The system file table is full.
ENOBUFS	Insufficient buffers were available in the system to complete the call.

Related Information

In this book: "gethostname, sethostname" in topic 1.2.100 and "rexec" in topic 1.2.235.

The discussions of **/etc/services**, **rlogind**, and **rshd** in *AIX TCP/IP User's*

Guide.

1.2.224 read, readv, readx

Purpose
Reads from a file or socket.

Syntax

```
#include <sys/uio.h>

int read (d, buf, nbyte)
int d;
char *buf;
unsigned int nbyte;
int readv (d, iov, iovcnt)
int d;
struct iovec *iov;
unsigned int iovcnt;

int readx (d, buf, nbyte, ext)
int d, ext;
char *buf;
unsigned int nbyte;
```

Description
The **read** system call reads a set number of bytes into a buffer. The **read** system call reads the number of bytes set by the **nbyte** parameter from the object associated with the **d** parameter and places those bytes into the buffer pointed to by the **buf** parameter.

The **readv** system call obtains data from the object associated with the **d** parameter and reads this data into the buffers specified by the array of **iovec** structures pointed to by the **iov** parameter.

The **d** parameter is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call, or a socket descriptor from a **socket** or **socketpair** system call.

The **iovec** structure is defined in the **sys/uio.h** header file, and it contains the following members:

```
caddr_t  iov_base;
int      iov_len;
```

Each **iovec** entry specifies the base address and length of an area in memory where data should be placed. The **readv** system call completely fills out an area before moving to the next.

On devices capable of seeking, the **read** starts at a position in the file given by the file pointer associated with the **d** parameter. Upon return from the **read** system call, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

When attempting to read from an empty pipe (or FIFO):

Unless **O_NDELAY** or **O_NONBLOCK** is set, the **read** blocks until data is written to the file or the file is no longer open for writing.

If **O_NDELAY** is set, the **read** returns 0.

If **O_NONBLOCK** is set, the **read** returns -1 and **errno** is set to EAGAIN.

When attempting to read a socket and no data is ready to be read:

Unless **O_NDELAY** or **O_NONBLOCK** is set, the **read** blocks until data becomes available.

If **O_NDELAY** is set, the **read** returns 0.

If **O_NONBLOCK** is set, the **read** returns -1 and **errno** is set to EAGAIN.

When attempting to read a file associated with a terminal that has no data currently available:

Unless **O_NDELAY** or **O_NONBLOCK** is set, the **read** blocks until data becomes available.

If **O_NDELAY** is set, the **read** returns 0.

If **O_NONBLOCK** is set, the **read** returns -1 and **errno** is set to EAGAIN.

When attempting to read a regular file that supports enforcement mode record locks, and all or part of the region to be read is currently locked by another process:

If **O_NDELAY** or **O_NONBLOCK** is set, then the **read** returns -1 and sets **errno** to EAGAIN.

If **O_NDELAY** and **O_NONBLOCK** are clear, then the **read** blocks the calling process until the lock is released.

For more information about record locks, see "fcntl, flock, lockf" in topic 1.2.78.

The **readx** system call performs the same function as **read**, except that it provides communication with character device drivers that require more information or return more status than **read** can handle.

For files, sockets, or special files with drivers that do not handle extended operations, the **readx** system call does exactly what the **read** system call does, and the **ext** parameter is ignored.

Each driver interprets the **ext** parameter in a device-dependent way, either as a value or as a pointer to a communication area. The nonextended **read** system call is equivalent to the extended **readx** system call with an **ext** parameter value of 0. Drivers must apply reasonable defaults when the **ext** parameter value is 0.

For directories, the **ext** parameter determines the format in which directory entries should be returned:

If the value of **ext** is 0 (implied by the **read** system call), the format in which directory entries are returned depends on the value of the **real directory read** flag (see "ulimit" in topic 1.2.313).

If the calling process does not have the **real directory read** flag set, the buffer specified by the **buf** parameter is filled with an array of directory entries truncated to fit the format of the **direct** structure (see "dir" in topic 2.3.16). This provides

compatibility with programs written for UNIX System V.

If the calling process has the **real directory read** flag set, the buffer specified by the **buf** parameter is filled with an image of the underlying implementation of the directory.

If the value of **ext** is 1, the buffer specified by the **buf** parameter is filled with consecutive directory entries in the format of a **dirent** structure (see "dir" in topic 2.3.16). This is used by the **readdir** library routine.

Other values of the **ext** parameter are reserved.

Note: On directories, the **read**, **readv**, and **readx** system calls start at the position specified by the file pointer associated with the **d** parameter. The value of this file pointer must either be 0 or a value which the file pointer had immediately after a previous call to **read** or **readx** on this directory. Upon return from the **read** or **readx** system call, the file pointer is incremented by a number which may not correspond to the number of bytes copied into the buffer.

Return Value

Upon successful completion for a file object, the **read**, **readv**, and **readx** system calls return the number of bytes actually read and placed in the buffer; this number may be less than the value of the **nbyte** parameter if the file is associated with a communication line, or if the number of bytes left in the file is less than the value of the **nbyte** parameter. A value of 0 is returned when an end-of-file has been reached. (For information about communication files, see "ioctlx, ioctl, gtty, stty" in topic 1.2.137 and "termio" in topic 2.5.28.)

For a socket object, the **read** system call returns the number of bytes actually read and placed into the buffer.

If the **read**, **readv**, or **readx** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions

The **read**, **readv**, and **readx** system calls fail if one or more of the following are true:

EBADF **d** is not a valid file descriptor open for reading or a valid socket descriptor.

EAGAIN An enforcement mode record lock is outstanding in the portion of the file that is to be read.

EFAULT **buf** points to a location outside of the process's allocated address space.

EDEADLK A deadlock would occur if the calling process were to sleep until the region to be read was unlocked.

EINTR A signal was caught during the **read** or **readv** system call.

EINVAL The value of **iovcnt** was not between 1 and 16, inclusive.

EINVAL One of the **iov_len** values in the **iov** array was negative.

EINVAL The sum of the **iov_len** values in the **iov** array overflowed a 32-bit integer.

EFAULT Part of the **iov** parameter points to a location outside of the process's allocated address space.

EAGAIN The object is marked for non-blocking I/O (**O_NONBLOCK**), and no data was ready to be read.

EINVAL An **nbyte** value of less than 0 is specified.

ENODEV The file specified is an invalid device for reading.

EIO A physical I/O error occurred.

EIO The process is in a background process group, attempting to read from its controlling terminal and either the process is ignoring or blocking the **SIGTTIN** signal or the process group of the calling process is orphaned.

If the Transparent Computing Facility is installed on your system, **read** or **readx** can also fail if one or more of the following are true:

ESITEDN1 The file cannot be read because a site went down and either no other copy of the file is available or the file is or was open for writing.

ESITEDN2 The operation was terminated because a site failed.

ENFILE The system inode table on another cluster site is out of space.

Related Information
In this book: "dup" in topic 1.2.64, "dup2" in topic 1.2.65, "fcntl, flock, lockf" in topic 1.2.78, "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "open, openx, creat" in topic 1.2.199, "pipe" in topic 1.2.204, "select" in topic 1.2.242, "socket" in topic 1.2.275, "socketpair" in topic 1.2.276, "fcntl.h" in topic 2.4.8, and "termio" in topic 2.5.28.

1.2.225 readlink

Purpose
Reads the value of a symbolic link.

Syntax

```
int readlink(path, buf, bufsiz)
char *path, *buf;
int bufsiz;
```

Description
The **readlink** system call places the first **bufsiz** characters of the contents of the symbolic link path into the user's buffer **buf**. Unless the link fills the buffer, it will be null-terminated.

Return Value
Upon successful completion, the count of characters place in the buffer, not including the terminating NULL, is returned to the calling process. Otherwise, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **readlink** system fails if one or more of the following are true:

EACCES **path** is not a symbolic link.

EFAULT **buf** points to an invalid address space.

EACCES A component of the path prefix denies search permission.

EIO A physical I/O error occurred.

ENOTDIR A component of the path prefix is not a directory.

ENOENT The named file does not exist.

ENOENT A null path name was provided.

ENOENT A hidden directory was named, but no component inside it matched the process's current site path list.

ENOENT A symbolic link was named, in the path prefix, but the file to which it refers does not exist.

EINVAL The file named by a **path** is not a symbolic link.

EFAULT **path** points outside the process's allocated address space.

ELOOP A loop of symbolic links was detected. Since **readlink** does not follow symbolic links in the last component of the path, this error cannot occur on the last component.

ENFILE The system inode table is full.

ENAMETOOLONG
A component of a path name exceeded 255 characters, or an entire path name exceeded 1023 characters.

If the Transparent Computing Facility is installed on your system,

readlink can also fail if one or more of the following are true:

ESITEDN1 **path** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE **path** is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of **path** is replicated but not stored on any site which is currently up.

EINTR A signal was caught during the system call.

Related Information
In this book: "stat.h" in topic 2.4.22 and "symlink" in topic 1.2.294.

1.2.226 reboot

Purpose
Reinitializes or halts system operation.

Syntax

```
#include <sys/reboot.h>

reboot(howto, dev)
int howto;
char *dev;
```

Description
The **reboot** system call requests that the system be reinitialized (rebooted) or terminated (halted).

The value of **howto** is interpreted as flag bits. The following flags are supported by AIX. Caller must have superuser privileges. If the **reboot** call fails, it returns to caller; otherwise, the call does not return.

RB_NOSYNC prevents the normal write of buffered data to file systems. If **RB_NOSYNC** is not on, all buffered file writes are completed and all file systems are unmounted and marked as clean. Cleanly unmounted file systems are not normally checked during system initialization.

RB_HALT causes the system to be halted instead of rebooted. If **RB_HALT** is specified, the AIX system is terminated and not restarted. If **RB_HALT** is omitted, the AIX system is terminated and immediately reinitialized.

The *dev* parameter is currently ignored.

Error Conditions

EPERM The calling process does not have **INSTALL_SYS** system privilege.

Related Information
The **reboot** command in *AIX Operating System Commands Reference*.

1.2.227 recv, recvfrom, recvmsg

Purpose
Receives a message from a socket.

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>

int recv (s, buf, len, flagsint recvfrom (s, buf, len, flags,
int s;                      int s;
char *buf;                  char *buf;
int len, flags;             int len, flags;
                            struct sockaddr *from;
int recvmsg (s, msg, flags) int *fromlen;
int s;
struct msghdr msg [ ];
int flags;
```

Description

The **recv** system call is normally used only on a connected socket (see "connect" in topic 1.2.49), but **recvfrom** and **recvmsg** can be used to receive data on a socket whether it is connected or not.

If the value of **from** is anything other than 0, the source address of the message is filled in. The **fromlen** parameter is initialized to the size of the buffer associated with the **from** parameter. On return, it is modified to indicate the actual size of the address stored there. These system calls return the length of the message. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from. For more information, see "socket" in topic 1.2.275.

If no messages are available at the socket, the receive system calls wait for a message to arrive, unless the socket is nonblocking. A socket marked nonblocking with the **O_NONBLOCK** flag or the **FIONBIO ioctl** returns a value of -1 and sets **errno** to EAGAIN in the situation where it would otherwise have blocked. A socket marked with the **O_NDELAY** flag returns ZERO in the situation where it would otherwise have blocked.

Use the **select** system call to determine when more data arrives. For more information, see "select" in topic 1.2.242.

The **flags** argument to a receive call is formed by logically ORing one or more of the values shown in the following list:

MSG_PEEK Peeks at incoming message.

MSG_OOB Processes out-of-band data. This flag can only be used for stream sockets in the INET domain. It is not supported in the UNIX domain and does not work for datagrams. In addition, only one byte of out-of-band data can be processed at a time.

The **recvmsg** system call uses a **msghdr** structure to minimize the number of directly supplied parameters. The **msghdr** structure is defined in the **sys/socket.h** header file, and it contains the following members:

```
caddr_t msg_name;          /* optional address              */
int     msg_namelen;       /* size of address               */
struct  iovec *msg_iov;    /* scatter/gather array          */
int     msg_iovlen;        /* # of elements in msg_iov      */
caddr_t msg_accrights;     /* access rights are currently   */
                           /* limited to file descriptors,  */
                           /* which each occupy the size    */
                           /* of an int           */
int     msg_accrightslen;  /* length of access rights       */
```

In the above structure, the fields are defined as follows:

msg_name Defines the destination address if the socket is unconnected. If no names are needed, you can use a NULL pointer for **msg_name**.

msg_namelen Specifies the size of **msg_name**.

msg_iov Describes the scatter gather locations.

msg_iovlen Specifies the number of elements in the **msg_iov** array.

msg_accrights Defines the access rights sent with the message.

msg_accrightslen Specifies the length of the access rights.

Return Value
Upon successful completion, the length of the message in bytes is returned. If the **recv**, **recvfrom**, or **recvmsg** system call fails, a value of -1 is returned, and **errno** is set to indicate the error. A socket marked with the **O_NDELAY** flag returns ZERO in the situation where it would otherwise have blocked.

Error Conditions
The system call fails if one or more of the following are true:

EBADF The **s** parameter is not valid.

ENOTSOCK The **s** parameter refers to a file, not a socket.

EAGAIN The socket is marked nonblocking with the **O_NONBLOCK** flag or the **FIONBIO ioctl** in the situation where it would otherwise have blocked.

EINTR The receive was interrupted by delivery of a signal before any data was available for the receive.

EFAULT The data was specified to be received into a nonwritable part of the user address space.

EINVAL The **msg_iovlen** field of the **msghdr** structure passed into a **recvmsg** call was negative.

Related Information
In this book: "send, sendto, sendmsg" in topic 1.2.246 and "socket" in topic 1.2.275.

1.2.228 regcmp, regex

Purpose
Compiles and matches regular-expression patterns.

Library
Programmers Workbench Library (**libPW.a**)

Syntax

```
char *regcmp (str [, str,...charc*regex (pat, subject [, ret,
char *str, *str,...;        char *pat, *subject, *ret,...;
                            extern char *__loc1;
```

Description
The **regcmp** subroutine compiles a regular expression (or pattern) and returns a pointer to the compiled form. The **str** parameters specify the pattern to be compiled. If more than one **str** parameter is given, then **regcmp** treats them as if they were concatenated together. It returns a NULL pointer if it encounters an incorrect parameter.

You can use the **regcmp** command to compile regular expressions into your C program, frequently eliminating the need to call the **regcmp** subroutine at run time.

The **regex** subroutine compares a compiled pattern to the **subject** string. Additional parameters are used to receive values. Upon successful completion, the **regex** subroutine returns a pointer to the next unmatched character. If the **regex** subroutine fails, a NULL pointer is returned. A global character pointer, **__loc1**, points to where the match began.

The **regcmp** and **regex** subroutines are borrowed from the **ed** command; however, the syntax and semantics have been changed slightly. You can use the following symbols with the **regcmp** and **regex** subroutines:

[ ] * . ^
These symbols have the same meaning as they do in the **ed** command.

- For **regex**, the minus within brackets means "through" according to the current collating sequence. For example, depending on the default collating sequence, **[a-z]** can be equivalent to **[abcd**...**xyz]** or **[aBbCc**...**xYyZz]**. You can use the **-** by itself if the **-** is the last or first character. For example, the character class expression **[]-]** matches the **]** (right bracket) and **-** (minus) characters.

$ Matches the end of the string. Use **\n** to match a new-line character.

+ A regular expression followed by **+** means one or more times. For example, **[0-9]+** is equivalent to **[0-9][0-9]***.

{m} {m,} {m,u}
Integer values enclosed in **{ }** indicate the number of times to apply the preceding regular expression. **m** is the minimum number and **u** is the maximum number. **u** must be less than 256. If you specify only **m**, it indicates the exact number of times to apply the regular expression. **{m,}** is equivalent to **{m**,&infinity.**}** and matches **m** or more occurrences of the expression. The plus **+** (plus) and ***** (asterisk) operations are equivalent to **{1,}** and **{0,}**, respectively.

(...)$n

This stores the value matched by the enclosed regular expression in the (**n**+1) (th) **ret** parameter. Ten enclosed regular expressions are allowed. **regex** makes the assignments unconditionally.

(...)
Parentheses group subexpressions. An operator, such as *****, **+**, or **{ }** works on a single character or on a regular expression enclosed in parenthesis. For example, **(a*(cb+)*)$0**.

All of the above defined symbols are special. You must precede them with a **** (backslash) if you want to match the special symbol itself. For example, **\$** matches a dollar sign.

The following special symbols are defined for internationalized regular expressions. Each is valid only within a range expression, (that is, between brackets).

[:alnum:]
Matches any alphanumeric, as defined by the **NLctype.h** macro **iswalnum**.

[:alpha:]
Matches any alpha, like **iswalpha**.

[:digit:]
Matches any digit, like **iswdigit**.

[:lower:]
Matches any lower, like **iswlower**.

[:print:]
Matches any printable, like **iswprint**.

[:punct:]
Matches any punctuation, like **iswpunct**.

[:space:]
Matches any white space, like **iswspace**.

[:upper:]
Matches any upper case letter, like **iswupper**.

[:xdigit:]
Matches any hex digit, like **iswxdigit**.

[=X=]
matches any character in the same equivalence class as **X**, as defined by **wceqvmap**.

[.XY.]
Matches the multiple character collating sequence XY as a single character (as defined by **_wcxcol**. For example, some Latin languages collate the sequence **ch** as a single character which falls between the letters c and d. The regular expression **[c[.ch.]d]amp** would match the words **camp**, **champ**, and **damp**.

The **ctype** sequences, such as **[:alpha:]**, cannot be used as end points of a range.

Note: **regcmp** uses the **malloc** subroutine to make the space for the vector. Always free the vectors that are not required. If you do not free

the unrequired vectors, you may run out of memory if **regcmp** is called repeatedly. Use the following as a replacement for **malloc** to reuse the same vector, thus saving time and space:

```
/* ...Your Program... */

malloc(n)
   int n;
{
   static int rebuf[256];

   return ((n <= sizeof(rebuf)) ? rebuf : NULL);
}
```

Examples

1. To perform a simple match:

```
char *cursor, *newcursor, *ptr;
...
newcursor = regex((ptr = regcmp("^\n", 0)), cursor);
free(ptr);
```

   This matches a leading new-line character in the subject string pointed to by **cursor**.

2. To extract a substring that matches a pattern:

```
char ret0[9];
char *newcursor, *name;
...
name = regcmp("([A-Za-z][A-Za-z0-9]{0,7})$0", 0);
newcursor = regex(name, "123Testing321", ret0);
```

   This matches the eight-character identifier **Testing3** and returns the address of the character after the last matched character (which is stored in **newcursor**). The string **Testing3** is copied into the character array **ret0**.

Related Information

In this book: "malloc, free, realloc, calloc, valloc, alloca, mallopt, mallinfo" in topic 1.2.162, "NCcollate, NCcoluniq, NCeqvmap, _NCxcol, _NLxcol" in topic 1.2.182, "wc_collate, wc_coluniq, wc_eqvmap, _wcxcol, _mbxcol, _wcxcolu, _mbxcolu" in topic 1.2.329, "setlocale" in topic 1.2.251, and "regexp: compile, step, advance" in topic 1.2.230.

The **ed** and **regcmp** commands in *AIX Operating System Commands Reference*.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.229 regex: re_comp, re_exec

Purpose
Handles regular expressions.

Library
Berkeley Compatibility Library (**libbsd.a**)

Syntax

```
char *re_comp (s)
char *s;

int *re_exec (s)
char *s;
```

Description
The **re_comp** subroutine complies a string into an internal form suitable for pattern matching. The **re_exec** subroutine checks the argument string against the last string passed to **re_comp**.

The **re_comp** subroutine returns 0 if the string **s** was compiled successfully; otherwise a string containing an error message is returned. If **re_comp** is passed to 0 or a null string, it returns without changing the currently compiled regular expression.

The **re_exec** subroutine returns 1 if the string **s** matches the last compiled regular expression, 0 if the string **s** failed to match the last compiled regular expression, and -1 if the compiled regular expression was invalid (indicating an internal error).

The strings passed to both **re_comp** and **re_exec** may have trailing or embedded newline characters; they are terminated by nulls. The regular expressions recognized are described in the manual entry for the **ed** command, given the above difference.

Return Value
If an error occurs, **re_exec** returns a -1, while **re_comp** returns one of the following strings:

```
No previous regular expression.
Regular expression too long.
Unmatched \(.
Missing ]
Too many \(\) pairs.
Unmatched \).
```

Related Information
In this book: "NCcollate, NCcoluniq, NCeqvmap, _NCxcol, _NLxcol" in topic 1.2.182, "regcmp, regex" in topic 1.2.228, and "regexp: compile, step, advance" in topic 1.2.230.

The **ed**, **grep**, and **sed** commands in *AIX Operating System Commands Reference*.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

1.2.230 regexp: compile, step, advance

Purpose
Compiles and matches regular-expression patterns.

Library
None

Syntax

```
#define INIT             declarations
#define GETC( )          getc_code
#define PEEKC( )         peekc_code
#define UNGETC(c)        ungetc_code
#define RETURN(pointer)  return_code
#define ERROR(val)       error_code

#include <regexp.h>



              char *compile (instring, ep,intdstep (p1,)p2)
              char *instring, *ep, *endbufchar *string, *expbuf;
              int seof;
                                          int advance (lp, ep)
                                          char *string, *expbuf;
```

Description
The **regexp.h** header file defines several general purpose subroutines that perform regular-expression pattern matching. Programs that perform regular-expression pattern matching such as **ed**, **sed**, **grep**, **bs**, and **expr** use this source file. In this way, only this file needs to be changed in order to maintain regular expression compatibility between programs.

The **NLregexp.h** functions compile, step and advance operate on file code strings. The following macros must be defined by the programmer prior to including **NLregexp.h**.

INIT
This macro is used for dependent declarations and initializations. It is placed right after the declaration and opening **{** (left brace) of the **compile** subroutine. The definition of **INIT** must end with a **;** (semicolon). **INIT** is frequently used to set a register variable to point the beginning of the regular expression so that this register variable can be used in the declarations for **GETC**, **PEEKC**, and **UNGETC**. Otherwise, you can use **INIT** to declare external variables that **GETC**, **PEEKC**, and **UNGETC** need.

```
#define INIT            register char *sp = instring; \
                        int sp_len; \
                        mbchar_t sp_peekc;
```

GETC( )
This macro returns the value of the next character (as an **mbchar_t**) in the regular expression pattern. Successive calls to the **GETC** macro should return successive characters of the pattern.

```
# define GETC()         (PEEK(),sp+=sp_len,sp_peekc)
```

PEEKC( )
This macro returns the next character (as an **mbchar_t**) in the regular expression. Successive calls to the **PEEKC** macro should return the same character, which should also be the next character returned by the **GETC** macro. The special value **ERR** should be returned if there is an error in the character.

```
#define PEEKC()  ( (-1==(sp_len=mbstomb (&sp_peekc,sp,MB_LEN_MAX) ) ) \
                        ?          sp_peekc=ERR\
                        :          sp_peekc)
```

UNGETC(c)
This macro causes the parameter **c** to be returned by the next call to the **GETC** and **PEEKC** macros. No more than one character of pushback is ever needed and this character is guaranteed to be that last character read by the **GETC** macro. The return value of the **UNGETC** macro is always ignored.

```
#define UNGETC (c)       (sp-=sp_len)
```

RETURN(pointer)
This macro is used on normal exit of the **compile** subroutine. The **pointer** parameter points to the first character immediately following the compiled regular expression. This is useful to programs that have memory allocation to manage.

```
#define RETURN(p)        return
```

ERROR(val)
This macro is used on abnormal exit from the **compile** subroutine. It should **never** contain a **return** statement. The **val** parameter is an error number. The error values and their meanings are:

```
#define ERROR(c)         regerr (c)
```

Error Name	Value	Meaning
BIG_RANGE	**11**	Range endpoint too large.
BAD_NUM	**16**	Bad number.
BAD_BACK	**25**	"\" digit out of range.
BAD_DELIM	**36**	Illegal or missing delimiter.
NO_SAVED	**41**	No remembered search string.
BAD_LEFTP	**42**	"\(\)" imbalance.
BAD_RIGHTP	**43**	Too many "\(".
EX_COMMA	**44**	More than two numbers given in \{ \}.
NO_CLOSE	**45**	"}" expected after "\".
MAX_MIN	**46**	First number exceeds second in \{ \}.
BAD_BRAK	**49**	"[ ]" imbalance.

TOO_BIG	**50**	Regular expression overflow.
STACK_EMPTY	**51**	Backtrack stack empty.
STACK_FULL	**52**	Backtrack stack full.
BAD_CHAR	**60**	Strange multibyte character.

The **compile** subroutine compiles the regular expression for later use. The **instring** parameter is never used explicitly by the **compile** subroutine, but you can use it in your macros. For instance, you may want to pass the string containing the pattern as the **instring** parameter to **compile** and use the **INIT** macro to set a pointer to the beginning of this string. (The following example uses this technique.) If your macros do not use **instring**, then call **compile** with a value of **((char *) 0)** for this parameter.

The **expbuf** parameter points to a character array where the compiled regular expression is to be placed. The **endbuf** parameter points to the location that immediately follows the character array where the compiled regular expression is to be placed. If the compiled expression cannot fit in (**endbuf-expbuf**) bytes, the call **ERROR(50)** is made.

The **eof** parameter is the character that marks the end of the regular expression. For example, in **ed** this character is usually **'/'** (slash).

The **regexp.h** header file defines other subroutines that perform actual regular-expression pattern matching. One of these is the **step** subroutine.

The **string** parameter of **step** is a pointer to a null-terminated string of characters to be checked for a match.

The **expbuf** parameter points to the compiled regular expression, which was obtained by a call to the **compile** subroutine.

The **step** subroutine returns the value 1 if the given string matches the pattern, and 0 if it does not match. If it matches, then **step** also sets two global character pointers: **loc1**, which points to the first character that matches the pattern, and **loc2**, which points to the character immediately following the last character that matches the pattern. Thus, if the regular expression matches the entire string, then **loc1** points to the first character of **string** and **loc2** points to the null character at the end of **string**.

The **step** subroutine uses the global variable **circf**, which is set by **compile** if the regular expression begins with a **^** (circumflex). If this variable is set, then **step** only tries to match the regular expression to the beginning of the string. If you compile more than one regular expression before executing the first one, then save the value of **circf** for each compiled expression and set **circf** to that saved value before each call to **step**.

The **step** subroutine calls a subroutine named **advance** with the same parameters that it was passed. The **step** function increments through the **string** parameter and calls **advance** until **advance** returns a 1, indicating a match, or until the end of **string** is reached. To constrain **string** to the beginning of the string in all cases, call the **advance** subroutine directly instead of calling **step**.

When **advance** encounters an ***** (asterisk) or a **\{ \}** sequence in the regular expression, it advances its pointer to the string to be matched as far as possible and recursively calls itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, **advance** backs up along the string until it finds a match or reaches the point in the string that initially matched the ***** or **\{ \}**. It is sometimes desirable to stop this backing-up before the initial point in the string is reached. If the global character pointer **locs** is equal to the point in the string sometime during the backing up process, **advance** breaks out of the loop that backs up and returns 0. This is used by **ed** and **sed** for global substitutions on the whole line so that expressions like **s/y*//g** do not loop forever.

Example
The following is an example of the regular expression macros and calls from the **grep** command.

```
#define INIT          register char *sp=instring;
#define GETC()        (*sp++)
#define PEEKC()       (*sp)
#define UNGETC(c)     (--sp)
#define RETURN(c)     return;
#define ERROR(c)      regerr()

#include <regexp.h>
...
compile (patstr, expbuf, &expbuf[ESIZE], '\0');
...
if (step (linebuf, expbuf))
   succeed ( );
...
```

Related Information
In this book: "NCcollate, NCcoluniq, NCeqvmap, _NCxcol, _NLxcol" in topic 1.2.182 and "regcmp, regex" in topic 1.2.228.

The **ed**, **grep**, and **sed** commands in *AIX Operating System Commands Reference*.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.231 Remote Procedure Call (RPC)

Purpose
Allows machines to make procedure calls to other network machines.

Library
Standard C Library (**libc.a**)

Description
Remote Procedure Call (RPC) is a remote procedure call specification that provides a procedure-oriented interface to remote services. RPC is used in networks to provide programs that enable communication between machines. For example, a network file service can be composed of programs that deal with high-level applications such as file access control and programs that deal with low-level applications such as **read** or **write**. The programs are accessible through a machine designated as a network server. A client of the network file service can call the procedures associated with the programs on behalf of a user logged in to the client machine.

A ***client*** is a computer or process that accesses the services or resources of another process or computer on the network. A ***server*** is a computer that provides services and resources, as well as implements network services. Each network service is a collection of remote programs. A remote program implements remote procedures. The procedures, along with their parameters and results, are documented in the specific program's protocol specification. A server can support more than one version of a remote program in order to be compatible with changing protocols.

In RPC, each server supplies a program that is a set of procedures. The combination of a host address, a program number, and a procedure number specifies one remote service procedure.

The RPC Communication Paradigm: Programs that communicate over a network need a paradigm for communication. The RPC paradigm is based on the remote procedure call model, which is similar to the local procedure call model. A local procedure call involves the caller placing arguments to a procedure in a defined location, such as a result register, and transferring control to the procedure. The caller eventually gains back control and extracts the results of the procedure from the defined location before continuing execution.

A remote procedure call is similar, except that one thread of control winds through two processes: a caller process and a server process. That is, the caller process sends a call message to the server process and waits for (or blocks) a reply message. The call message contains information that includes the parameters of the procedure. The reply message contains information that includes the results of the procedure. When the caller receives the reply message, it extracts the results of the procedure and resumes execution.

On the server side, a process is dormant awaiting the arrival of a call message. When one arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message, and waits for the next call message.

Only one of the two processes is active at any given time. That is, the RPC protocol does not explicitly support multithreading of caller or server processes.

External Data Representation (XDR): RPC uses External Data Representation

(XDR) to establish uniform representations for data types in order to transfer the call message data between machines without regard to their manufacturers, operating systems, or architectures. For basic data types such as integers and strings, XDR provides primitives that serialize, or translate, information from the local host's representation to XDR's representation, and deserialize, or translate, from the XDR representation to the local host's representation. XDR also uses constructor primitives that allow the use of the basic data types to create more complex data types, such as arrays and discriminated unions.

RPC input and output data structures are described using XDR's data description language, which resembles the C programming language. Many of the constructs are identical to those in the C language. XDR additionally uses a construct called a ***discriminated union***. The discriminated union is a union data structure which holds various objects with one of the objects identified directly by a discriminant, or arm, that is the first item to be serialized or deserialized.

The syntax for XDR routines that are called directly by RPC routines are included in "RPC Subroutines" in topic 1.2.231.2.5. For more detailed information about XDR and the other XDR routines, see "XDR (External Data Representation)" in topic 1.2.332.

Data Transports and Semantics: RPC deals with the specification and interpretation of messages, not with the method used to pass messages from one process to the other. It does not depend on services provided by specific transport protocols. Although specific semantics, or meanings, are not attached to remote procedures or their execution, certain semantics can be inferred from the protocol of the underlying data transport that is used.

For example, passing RPC messages with the UDP/IP data transport is unreliable. If the caller retransmits RPC call messages after short timeouts, the only thing it can infer from no reply message is that the remote procedure was executed zero or more times (and from a reply message, one or more times). In contrast, passing RPC messages with TCP/IP is reliable. No reply message means the remote procedure was executed one time at most, and a reply message means that the remote procedure was executed exactly once.

Binding and Rendezvous Independence: RPC does not bind a client to a service as part of its protocol. This required function is left up to a higher level software. However, the network software can use RPC to accomplish the tasks involved with binding clients to services.

Message Authentication: The RPC protocol provides the fields required for a client to identify itself to a service and for a service to identify itself to the client. The contents of RPC authentication parameters for these fields are determined by the type, sometimes called flavor, of the authentication used by the server and the client. A server can support multiple types of authentication at one time.

You can build additional security and access controls on top of the message authentication.

Subtopics
1.2.231.1 The RPC Protocol
1.2.231.2 The RPC Message Protocol

1.2.231.1 The RPC Protocol

RPC is primarily a tool for calling remote procedures. By providing a unique specification for calling the remote procedures, RPC can match a reply message to each request (or call) message.

Each RPC call message contains the following unsigned fields to uniquely identify the procedure to be called:

- Remote program numbe
- Remote program version numbe
- Remote procedure number

Assigning Program Numbers to Protocols: Program numbers are assigned in groups of 0x20000000 (536870912) as shown in Figure 2-2:

Figure 2-2. How to Assign Program Numbers

Program Number	**How Assigned**	**Use**
0 - 1fffffff	Defined by system authority	System authority (the product licensor) administers this first group of numbers. This group should be identical for all system customers.
20000000 - 3fffffff	Defined by user	Use this group for applications you develop and for debugging new programs.
40000000 - 5fffffff	Transient	Use the third group for applications that generate program numbers dynamically.

Assigning Version Numbers to Programs: As programs evolve into more stable and mature protocols, version numbers are assigned. The first implementation of a remote program is usually designated as version number 1 (or a similar form).

The version number identifies which version of the protocol the caller is using. Version numbers make it possible to use old and new protocols through the same server.

Assigning Procedure Numbers to Programs: The procedure numbers are documented in each program's protocol specification. For example, a file service protocol's specification can list the **read** procedure as procedure number **5** and **write** as procedure number **12**.

1.2.231.2 The RPC Message Protocol

The RPC message protocol consists of two distinct forms: the call message and the reply message. A client makes a remote procedure call to a network server and receives a reply containing the results of the procedure's execution. RPC message protocols are defined in the XDR data description language.

Call and reply messages have the following values:

```
  enum msg_type {
          CALL = 0,
          REPLY = 1
  };
```

Subtopics
1.2.231.2.1 Message Protocol Structure
1.2.231.2.2 Record Marking in the Messages
1.2.231.2.3 Authentication
1.2.231.2.4 The Portmap Program
1.2.231.2.5 RPC Subroutines

1.2.231.2.1 Message Protocol Structure

The initial message structure appears as follows:

```
struct rpc_msg  {
    unsigned            xid;
    union switch (enum msg_type)  {
          CALL:    struct call_body;
          REPLY;   struct reply_body;
    };
};
```

Both the RPC call and reply messages start with an **xid**, the transaction identifier, followed the two-armed discriminated union **enum msg_type**. The reply messages' **xid**s are matched to the **xid**s of the call messages. It is important to note that the **xid**s are used only by the clients when matching reply messages to call messages.

The initial structure is followed by the body of the message. The body of a call message has a single form. The body of a reply message takes one of two forms depending on whether a call is accepted or rejected by the server.

Call Messages: The body of an RPC call message is structured as follows:

```
struct call_body  {
     unsigned rpcvers;
     unsigned prog;
     unsigned vers;
     unsigned proc;
struct opaque_auth cred;
struct opaque_auth verf;
#1 parameter
#2 parameter...
};
```

The structure is explained in the following:

rpcvers RPC protocol specification version number.

prog Number that identifies a remote program. This is an assigned number represented in a protocol that identifies the program needed to call a remote procedure. Program numbers are administered by a central authority and are documented in the program's protocol specification.

vers Number that identifies the remote program's version. As a remote program's protocols are implemented, they evolve and change. Version numbers are assigned to identify different stages of a protocol's evolution. Servers can service requests for different versions of the same protocol simultaneously.

proc Number of the procedure associated with the remote program being called. These numbers are documented in the specific program's protocol specification. For example, a protocol's specification can list the **read** procedure as procedure number **5** or **write** as procedure number **12**.

cred Credentials authentication parameter that identifies the caller

as having permission to call the remote program. It is passed as an opaque data structure, which means the data is not interpreted as it is passed from the client to the server.

verf Verifier authentication parameter that identifies the caller to the server. It is passed as an opaque data structure which means the data is not interpreted as it is passed from the client to the server.

Note: Procedure specific parameters appear at the end of the call body.

Reply Messages: RPC reply messages take one of two forms to show that the call message was accepted by a network server or that it was rejected by the server. The discriminant **enum reply_stat** acts as a switch to the rejected or accepted reply message form.

```
enum reply_stat {
          MSG_ACCEPTED = 0,
          MSG_DENIED = 1
};
```

A reply to an RPC request accepted by the network server takes the following form:

```
struct accepted_reply {
     struct opaque_auth  verf;
     union switch (enum accept_stat) {
           SUCCESS: struct {
           return values
           };
           PROG_MISMATCH: struct {
                unsigned low;
                unsigned high;
           };
           default:  struct {
           };
     };
};
```

The structures in the accepted reply are specified by the following:

opaque_auth verf;
Authentication verifier generated by the server to identify itself to the caller.

enum accept_stat
A discriminant that acts as a switch to one of the following structures:

SUCCESS
Defines the results or return values of the procedure.

PROG_MISMATCH
Specifies the lowest and highest version numbers of the remote program that are supported by the server.

default
Lists errors that occur even though the request was accepted.

The **default** structure can take any of the following values:

PROG_UNAVAIL
The remote server has not exported the program.

PROG_MISMATCH
The remote server cannot support the client's version number.

PROC_UNAVAIL
The program cannot support the requested procedure.

GARBAGE_ARGS
The procedure cannot decode the parameters specified in the call.

A reply to an RPC request that is rejected by the server takes the following form:

```
struct rejected_reply {
     union switch (enum reject_stat) {
           RPC_MISMATCH: struct {
                unsigned low;
                unsigned high;
           };
           AUTH_ERROR:  enum auth_stat;
     };
};
```

The discriminant, **enum reject_stat**, acts as a switch to one of the following:

RPC_MISMATCH
The server is not running a compatible version of the RPC protocol. The server returns the lowest and highest version numbers available.

AUTH_ERROR
The server refuses to authenticate the caller and returns a failure status with the value **enum auth_stat**. The **enum auth_stat** status returned is one of the following:

AUTH_BADCRED
The caller had bad credentials.

AUTH_REJECTEDCRED
The client must begin a new session.

AUTH_BADVERF
The clients verifier was bad.

AUTH_REJECTEDVERF
The verifier expired or replayed.

AUTH_TOOWEAK
The authentication credentials were rejected for security reasons.

Multiple Reply Messages: A client can broadcast its message across the network and wait for many replies by using broadcast RPC with the UDP/IP transport. Servers that support broadcast protocols only respond when the request is successfully processed. Otherwise they are silent.

No Reply Message Needed: In cases where the client does not need a reply, RPC can batch, or send, a large sequence of call messages to a server using the TCP/IP transport. The batch sequence is terminated by another RPC that is called to clear the pipeline.

1.2.231.2.2 Record Marking in the Messages

When RPC messages are passed using the TCP/IP byte stream protocol for data transport, it is important to identify the end of one message and the start of the next one by record marking (**rm**).

A record is composed of one or more record fragments. A record fragment is a 4-byte header. The header is followed by **0** to **2(32)-1** bytes of fragment data. The bytes encode an unsigned binary number, similar to XDR integers. The order of bytes is from highest to lowest. This binary number encodes a Boolean and an unsigned binary value of 31 bits.

The Boolean value is the highest-order bit of the header. If the Boolean value is value is 1, the fragment is the last fragment of the record. The unsigned binary value is the length in bytes of the fragment's data.

1.2.231.2.3 Authentication

RPC provides the **opaque_auth** structure for the verifier parameter so that a client can identify itself to the server receiving the call message, and for the server to identify itself to the client in the response message.

In addition, RPC provides the **auth_unix** structure for the credentials parameter so that client access permission to the remote program can be checked by the server. These RPC authentication parameters are opaque data type structures. That is, they pass through the messages without being interpreted. See the "Opaque Data" in topic 1.2.332.1.1 for more detailed information about this data type.

If neither the caller or the server require permission checking, **AUTH_NULL** can be used for the RPC message credentials and verifier parameters. These routines are discussed in alphabetical order in "RPC Subroutines" in topic 1.2.231.2.5.

AIX handles the structure of the credentials parameter in a shorthand form. The caller of a remote procedure can use this shorthand representation by using **AUTH_UNIX** as the value of the credentials parameter. The bytes of the credentials string encode the following XDR structure:

```
struct auth_unix {
          unsigned        stamp;
          string          machinename<255>;
          unsigned        uid;
          unsigned        gid;
          unsigned        gids<10>;
};
```

The parameters in the structure are defined as follows:

stamp	Arbitrary ID generated by the caller's machine.
machinename<255>	Name of the caller's machine. The name must not exceed 255 bytes in length.
uid	Caller's effective user ID.
gid	Caller's effective group ID.
gids<10>	Counted array of groups which contain the caller as a member. A maximum of 10 groups is allowed.

The value of the response verifier's discriminant in the reply message (**oa_flavor** in the **opaque_auth** structure) from the server is either **AUTH_NULL** or **AUTH_SHORT**. If the value is **AUTH_SHORT**, the bytes of the **verf** string encode an **auth_opaque** structure. The **auth_opaque** structure can then be passed to the server in place of the original **AUTH_UNIX** credentials. The server keeps a cache that maps the **auth_opque** structures to the credentials of the caller. The caller requires less network bandwidth and server CPU time when the shorthand credentials are used.

Note: The server can eliminate, or flush, the shorthand **auth_opaque** structures at any time. If this happens, an RPC message's rejection is listed as an authentication error. The original

AUTH_UNIX must be used to generate the shorthand version again.

1.2.231.2.4 The Portmap Program

RPC uses a **portmap** daemon, also known as the **portmapper**, to map the RPC program version numbers to UDP/IP or TCP/IP port numbers. This maximizes the efficiency of remote program bindings since the range of reserved port numbers is small compared to the number of remote programs possible. The **portmap** daemon runs on a reserved port, answering client queries regarding the location of the port numbers of the remote programs.

1.2.231.2.5 RPC Subroutines

The RPC subroutines are listed alphabetically in this section. Each subroutine is introduced by its syntax and followed by a brief discussion of its purpose, parameters, and return value.

A Note about the Parameters: RPC uses the following common parameters to identify the remote procedure called in each routine:

prognum Number of the remote program. Program numbers are administered by a central authority.

versnum Version number of the remote program. As remote program's protocols are implemented, they evolve and change. Version numbers are assigned to identify different stages of the protocols evolution. Servers can service requests for different versions of the same protocol simultaneously.

procnum Procedure numbers that identify the procedure to be called. These numbers are documented in the specific program's protocol specification. For example, a protocol's specification can list the **read** procedure as procedure number **5** or **write** as procedure number **12**.

Note: The structure of these parameters is discussed in "Call Messages" in topic 1.2.231.2.1.

Other parameters specific to the routines are identified in the discussion of each routine.

void
auth_destroy (auth)
AUTH *auth;

The **auth_destroy** macro destroys the authentication information structure pointed to by the **auth** parameter. Destroying the structure deallocates private data structures associated with it. The use of **auth** is undefined after calling this macro.

AUTH *
authnone_create ( )

The **authnone_create** subroutine creates and returns an RPC authentication handle that passes no usable authentication with each remote procedure call.

AUTH *
authunix_create (host, uid, gid, len, aup_gids)
char *host;
int uid, gid, len;
int *aup_gids;

The **authunix_create** subroutine creates and returns an RPC authentication handle with AIX permissions.

The **host** parameter points to the name of the machine on which the permissions were created. The **uid** parameter specifies the user's user ID.

The **gid** parameter specifies the current group ID of the user, while the **aup_gids** parameter points to the counted array of groups to which the user belongs. The length of the groups array is specified by the **len** parameter.

AUTH *
void authunix_create_default ( )

The **authunix_create_default** subroutine calls the **authunix_create** subroutine to create and return the default AIX authentication handle.

callrpc (host, prognum, versnum, procnum, inproc, in, outproc, out)
char *host;
u_long prognum, versnum, procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;

The **callrpc** subroutine calls a remote procedure associated with the **prognum**, **versnum**, and **procnum** parameters on the machine pointed to by the **host** parameter.

The **inproc** parameter specifies the procedure that encodes the procedure's parameters. The **in** parameter points to the address of the procedure's arguments.

The **outproc** parameter specifies the procedure that decodes the procedure's results. The **out** parameter points to the address where results are placed.

Upon successful completion, this routine returns the value 0. Otherwise, a nonzero value of the type **enum clnt_stat** is returned.

Note: Calling remote procedures with this routine uses UDP/IP as a transport. If the server is a TCP/IP supported server only, you cannot get a connection. See the **clnttcp_create** subroutine to open TCP/IP sockets.

enum clnt_stat
clnt_broadcast (prognum, versnum, procnum, inproc, in, outproc, out, eachresul
u_long prognum, versnum, procnum;
xdrproc_t inproc;
char *in;

```
xdrproc_t outproc;
char *out;
resultproc_t eachresult;
```

The **clnt_broadcast** subroutine broadcasts a remote procedure call to all locally connected networks.

The remote procedure is identified by the **prognum**, **versnum**, and **procnum** parameters on the machine identified by the parameter **host**.

The **inproc** parameter specifies the procedure that encodes the procedure's parameters. The **in** parameter points to the address of the procedure's arguments.

The **outproc** parameter specifies the procedure that decodes the procedure's results. The **out** parameter points to the address where results are placed.

When a client broadcasts a remote procedure call over the network, a number of server processes respond. Each time it receives a response, this routine calls the **eachresult** routine to point to the function that is called. The **eachresult** routine takes the following form:

```
eachresult (out, addr)
char *out;
struct sockaddr_in *addr;
```

The **out** parameter points to the address where the procedure's results are decoded and placed. The **addr** parameter points to the address of the machine that sent the results.

If **eachresult** returns 0, the **clnt_broadcast** subroutine waits for more replies. Otherwise, it returns with the appropriate results.

```
enum clnt_stat
clnt_call  (clnt, procnum; inproc, in, outproc, out, tout)
CLIENT *clnt;
long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
struct timeval tout;
```

The **clnt_call** macro calls the remote procedure associated with a client handle pointed to by the **clnt** parameter. The **clnt** parameter is the result of an RPC client creation routine, such as **clntudp_create**, which opens a UDP/IP socket.

The **procnum** parameter identifies the remote procedure on the host machine associated with the client handle.

The **inproc** parameter specifies the procedure that encodes the procedure's parameters. The **in** parameter points to the address of the procedure's arguments.

The **outproc** parameter specifies the procedure that decodes the procedure's results. The **out** parameter points to the address where results are placed. The **tout** parameter sets the time allowed for results to come back.

clnt_destroy (clnt)
CLIENT *clnt;

The **clnt_destroy** macro destroys the client's RPC handle. The **clnt** parameter is the result of an RPC client creation routine, such as **clntudp_create** which opens a UDP/IP socket. Destroying the client's RPC handle deallocates private data structures, including the **clnt** structure itself. The use of **clnt** is undefined after calling the **clnt_destroy** macro.

The user must close the sockets associated with the **clnt** structure.

clnt_freeres (clnt, outproc, out)
CLIENT *clnt;
xdrpoc_t outproc;
char *out;

The **clnt_freeres** macro frees data that was allocated by the RPC/XDR system when it decoded the results of an RPC call.

The **clnt** parameter points to the structure of the client handle. The **outproc** parameter specifies the XDR routine that describes the results in simple decoding primitives. The **out** parameter points to the address where the results are placed.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

void
clnt_geterr (clnt, errp)
CLIENT *clnt;
struct rpc_err *errp;

The **clnt_geterr** macro copies error information from a client handle to an error structure.

The **clnt** parameter points to the client handle. The **errp** parameter points to the address of the error structure.

void
clnt_pcreateerror (s)
char *s;

The **clnt_pcreateerror** subroutine writes a message to standard error indicating why a client RPC handle could not be created. The **s** parameter points to a character string that represents the error text.

This subroutine is used after a **clntraw_create**, **clnttcp_create**, or **clntudp_create** call.

```
void
clnt_perrno (stat)
enum clnt_stat stat;
```

The **clnt_perrno** subroutine writes a message to standard error corresponding to the condition specified by the **stat** parameter.

This subroutine is used after a **callrpc** subroutine.

```
clnt_perror (clnt, s)
CLIENT *clnt;
char *s;
```

The **clnt_perror** subroutine writes a message to standard error indicating why a remote procedure call failed.

The **clnt** parameter is the client handle used to make the call. The **s** parameter points to a character string that represents the error text.

This routine is used after a **clnt_call** subroutine.

```
CLIENT *
clntraw_create (prognum, versnum)
u_long prognum, versnum;
```

The **clntraw_create** subroutine creates a toy RPC client for simulation of the remote program specified by the **prognum** and **versnum** parameters. The client uses a buffer located within the address space of the process as the transport to pass messages to the service. If the corresponding RPC server lives in the same address space, simulation of RPC and acquisition of RPC overheads, such as round-trip times, are done without kernel interference. (See the **svcraw_create** subroutine.)

On successful completion, this subroutine returns a pointer to a valid RPC client. If this subroutine fails, it returns the value NULL.

```
CLIENT *
```

```
clnttcp_create (addr, prognum, versnum, sockp, sendsz, recvsz)
struct sockaddr_in *addr;
u_long prognum, versnum;
int *sockp;
u_int sendsz, recvsz;
```

The **clnttcp_create** subroutine creates an RPC client for a remote program identified by the **prognum** and **versnum** parameters. The client uses TCP/IP as the transport to pass messages to the service.

The **addr** parameter points to the Internet address of the remote program. If port number for this Internet address (**addr->sin_port**) is 0, then **addr** is set to the actual port that the remote program is listening on. The client making the remote procedure call consults the remote **portmap** daemon for this information.

The **sockp** parameter is a pointer to a socket. If **sockp** is **RPC_ANYSOCK**, the **clnttcp_create** subroutine opens a new socket and sets the **sockp** pointer to it.

Since TCP/IP remote procedure calls use buffered I/O, users can set the size of the send and receive buffers with the **sendsz** and **recvsz** parameters. If the size of either buffer is set as 0, the subroutine picks suitable default values.

Upon successful completion, this routine returns a valid TCP/IP client. If it fails, it returns the value NULL.

```
CLIENT *
clntudp_create (addr, prognum, versnum, wait, sockp)
struct sockaddr_in *addr;
u_long prognum, versnum;
struct timeval wait;
int *sockp;
```

The **clntudp_create** subroutine creates an RPC client for a remote program identified by the **prognum** and **versnum** parameters. The client uses UDP/IP as the transport to pass messages to the service.

The **addr** parameter points to the Internet address of the remote program. If port number for this Interenet Address (**addr->sin_port**) is 0, then **addr** is set to the actual port that the remote program is listening on. The **clntudp_create** subroutine consults the remote **portmap** daemon for this information.

The **sockp** parameter is a pointer to a socket. If **sockp** is **RPC_ANYSOCK**, the **clntudp_create** subroutine opens a new socket and sets the **sockp** pointer to it.

The **wait** parameter sets the amount of time that the UDP/IP transport waits until a response is received before it resends the remote procedure call or the remote procedure call times out. The total time for the call to time out is set by the **clnt_call** subroutine.

Note: RPC messages transported by UDP/IP can hold up to 8K bytes of encoded data. Use this transport for procedures that take

arguments or return results of less than 8K bytes.

```
void
get_myaddress (addr)
struct sockaddr_in *addr;
```

The **get_myaddress** subroutine gets the machine's IP address without consulting the library routines that access the **/etc/hosts** file.

The **addr** parameter points to an address where the machine's IP address is placed. The port number is set to **htons(PMAPPORT)**.

```
struct pmaplist *
pmap_getmaps (addr)
struct sockaddr_in *addr;
```

The **pmap_getmaps** subroutine acts as the user interface to the **portmap** daemon to return a list of the current RPC program-to-port mappings on the host located at the IP address pointed to by the **addr** parameter.

This routine can return the value NULL.

Note: The command **rpcinfo -p** uses this routine. See *AIX Operating System Commands Reference* for more information about this command.

```
u_short
pmap_getport (addr, prognum, versnum, protocol)
struct sockaddr_in *addr;
u_long prognum, versnum, protocol;
```

The **pmap_getport** subroutine acts as the user interface to the **portmap** daemon to return the port number on which a service waits.

The **addr** parameter points to the IP address of the host where the remote program that supports the waiting service resides. The **prognum** and **versnum** parameters identify the remote program that supports the waiting service. The **protocol** parameter specifies the transport protocol that the service recognizes.

If the routine returns a value of 0, the mapping does not exist or the RPC system did not contact the remote **portmap** daemon. If the remote **portmap** daemon was not contacted, the **rpc_createerr** global variable contains the RPC status.

```
enum clnt_stat
pmap_rmtcall (addr, prognum, vernsum, procnum, inproc, in,
```

```
          outproc, out, tout, portp)
struct sockaddr_in *addr;
u_long prognum, versnum, procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
struct timeval tout;
u_long *portp;
```

The **pmap_rmtcall** subroutine instructs the **portmap** daemon on the host at the IP address pointed to by the **addr** parameter to make a remote procedure call on behalf of the caller to a procedure on that host. The **portp** parameter is modified to the program's port number if the procedure succeeds.

The **prognum**, **versnum**, and **procnum** parameters identify the program associated with the remote procedure.

The **inproc** parameter specifies the XDR routine that encodes the remote procedure's parameters. The **in** parameter points to the address of the procedure's arguments.

The **outproc** parameter specifies the XDR routine that decodes the remote procedure's results. The **tout** parameter sets the time the routine waits for the results to return before resending the call.

Notes:

1. Use this procedure for a **ping** command only. See *Interface Program for use with TCP/IP* for information on **ping**.

2. Also see the **clnt_broadcast** subroutine in this book.

```
pmap_set (prognum, versnum, protocol, port)
u_long prognum, versnum, protocol;
u_short port;
```

The **pmap_set** subroutine acts as a user interface to the **portmap** daemon to map a remote procedure to a port on the machine's **portmap** daemon. The port on the machine's **portmap** daemon is specified by the **port** parameter.

The **prognum**, **versnum**, and **protocol** parameters identify the remote procedure call. The values for the **protocol** parameter can be **IPPROTO_UDP** or **IPPROTO_TCP**.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

Note: The **pmap_set** subroutine is called by the **svc_register**.

```
pmap_unset (prognum, versnum)
u_long prognum, versnum;
```

The **pmap_unset** subroutine destroys mappings between the remote procedure call and the ports on the machine's **portmap** daemon. The **prognum** and **versnum** parameters identify the remote procedure call.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

```
registerrpc (prognum, versnum, procnum, procname, inproc, outproc)
u_long prognum versnum, procnum;
char * (*procname) ();
xdrproc_t inproc, outproc;
```

The **registerrpc** subroutine registers a procedure identified by the **procname** parameter with the RPC service package.

If a request arrives that matches the values of the **prognum**, **versnum**, and **procnum** parameters, **procname** is called with a pointer to its parameters, and returns a pointer to its static results.

The **inproc** parameter specifies the XDR routine that decodes the procedure's parameters. The **outproc** parameter specifies the XDR routine that encodes the procedure's results.

Upon successful completion, this routine returns the value 0. Otherwise, it returns the value of -1.

Note: Remote procedures registered in this form are accessed using the UDP/IP transport protocol only. See the **svcudp_create** subroutine for restrictions.

```
struct rpc_createerr  rpc_createerr;
```

The **rpc_createerr** global variable is set by any RPC client creation routine that does not succeed. Use the **clnt_pcreateerror** subroutine to write to standard output the reason why the client was not created.

```
svc_destroy (xprt)
SVCXPRT *xprt;
```

The **svc_destroy** macro destroys the RPC service transport handle pointed to by the **xprt** parameter. Destroying the handle involves deallocating the private data structures, including **xprt** itself. Use of **xprt** is undefined after calling this routine.

```
int svc_fds;
```

The **svc_fds** global variable reflects the RPC service's file

descriptor bit mask. This variable is used to manually run asynchronous processing in a program instead of calling the **svc_run** procedure that sets the asynchronous processing automatically.

This is a read-only variable, but its value can change as a result of the **svc_getargs** or a creation routine. Do not pass the address of this variable to the **select** system call.

svc_freeargs (xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;

The **svc_freeargs** macro frees data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using the **svc_getargs** routine. The **xprt** parameter points to the RPC service transport handle.

The **inproc** parameter specifies the XDR routine that decodes the arguments. The **in** parameter points to the address where the procedure's arguments are placed.

If this routine successfully frees the results, it returns the value TRUE. Otherwise, it returns the value FALSE.

svc_getargs (xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;

The **svc_getargs** macro decodes the arguments of an RPC request associated with the RPC service transport pointed to by the **xprt** parameter.

The **inproc** parameter specifies the XDR routine that decodes the arguments. The **in** parameter points to the address where the arguments are placed.

If this routine successfully frees the results, it returns the value TRUE. Otherwise, it returns the value FALSE.

struct sockaddr_in
svc_getcaller (xprt)
SVCXPRT *xprt;

The **svc_getcaller** subroutine gets the network address of the caller of a procedure associated with the RPC service transport handle that is pointed to by the **xprt** parameter.

svc_getreq (rdfds)
int rdfds;

The **svc_getreq** subroutine is called when the **select** system call has determined an RPC request to service the RPC sockets associated with the value of the **rdfds** parameter. The **rdfds** parameter is the read-file descriptor bit mask.

The **svc_getreq** subroutine returns when all associated sockets with the value **rdfds** have been serviced.

Note: This routine is used to manually set asynchronous event processing in a program instead of calling the **svc_run** to automatically set it.

svc_register (xprt, prognum, versnum, dispatch, protocol)
SVCXPRT *xprt;
u_long prognum, versnum;
void (*dispatch) ();
u_long protocol;

The **svc_register** subroutine maps a remote procedure with a service dispatch procedure pointed to by the **dispatch** parameter.

The **xprt** parameter points to an RPC service transport handle.

The **prognum** and **versnum** parameters identify the remote program associated with the remote procedure.

The **protocol** parameter specifies the data transport used by the service. If **protocol** is 0, the service is not registered with the **portmap** daemon. If **protocol** is not 0 (or is **IPPROTO_UDP** or **IPPROTO_TCP**), the remote procedure triple [**prognum**, **versnum**, and **protocol**] is mapped to the **xprt->xp_port** port.

The **dispatch** procedure takes the following form:

```
dispatch (request, xprt)
struct svc_req *request;
SVCXPRT *xprt;
```

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

svc_run ( )

The **svc_run** subroutine waits for RPC service requests to arrive. When a request arrives, **svc_run** calls the appropriate service procedure using the **svc_getreq** subroutine. This procedure is usually waiting for a **select** system call to return.

The **svc_run** subroutine never returns.

```
svc_sendreply (xprt, outproc, out)
SVCXPRT *xprt;
xdrproc_t outproc;
char *out;
```

The **svc_sendreply** subroutine sends back the results of a remote procedure call. This routine is called by an RPC service's dispatch routine.

The **xprt** parameter points to the RPC service transport handle of the caller. The **outproc** parameter specifies the XDR routine that encodes the results. The **out** parameter points to the address where results are placed.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

```
void
svc_unregister (prognum, versnum)
u_long prognum, versnum;
```

The **svc_unregister** subroutine removes mappings between procedures and objects. It removes the mapping of the service procedure identified by the **prognum** and **versnum** parameters to dispatch routines. It also removes the mapping of the service procedure identified by the **prognum** and **versnum** parameters to a port number.

```
void
svcerr_auth (xprt, why)
SVCXPRT *xprt;
enum auth_stat why;
```

The **svcerr_auth** subroutine is called by a service dispatch routine that refuses to perform a remote procedure call because of an authentication error.

The **xprt** parameter points to the RPC service transport handle. The **why** parameter specifies the authentication error.

```
void
svcerr_decode (xprt)
SVCXPRT *xprt;
```

The **svcerr_decode** subroutine is called by a service dispatch routine that cannot decode its parameters. The **xprt** parameter points to the RPC service transport handle.

Note: See the **svc_getargs** subroutine.

```
void
svcerr_noproc (xprt)
SVCXPRT *xprt;
```

The **svcerr_noproc** subroutine is called by a service dispatch routine that does not implement the procedure number the caller requested. The **xprt** parameter points to the RPC service transport handle.

```
void
svcerr_noprog (xprt)
SVCXPRT *xprt;
```

The **svcerr_noprog** subroutine is called when the requested program is not registered with the RPC package. The **xprt** parameter points to the RPC service transport handle.

Note: Service implementors do not usually need this routine.

```
void
svcerr_progvers (xprt)
SVCXPRT *xprt;
```

The **svcerr_progvers** subroutine is called when the requested version of a program is not registered with the RPC package. The **xprt** parameter points to the RPC service transport handle.

Note: Service implementors do not usually need this routine.

```
void
svcerr_systemerr (xprt)
SVCXPRT *xprt;
```

The **svcerr_systemerr** subroutine is called by a service dispatch routine when it detects a system error not covered by a protocol. For example, a service calls this routine if it can no longer allocate storage. The **xprt** parameter points to the RPC service transport handle.

```
void
svcerr_weakauth (xprt)
SVCXPRT *xprt;
```

The **svcerr_weakauth** subroutine is called by a service dispatch routine that cannot make the remote procedure call because the

supplied authentication parameters were insufficient. It is called even if the given parameters are correct. The **xprt** parameter points to the RPC service transport handle.

The **svcerr_weakauth** subroutine calls the **svc_auth** routine using the correct RPC service transport handle (**xprt**) and as the authentication error the argument **AUTH_TOOWEAK**.

SVCXPRT *
svcraw_create ( )

The **svcraw_create** subroutine creates a toy RPC service transport. The service transport is located within the address space of the process. If the corresponding RPC server resides in the same address space, simulation of RPC and acquisition of RPC overheads, such as round-trip times, are done without kernel interference. (See the **clntraw_create** routine on page 1.2.231.2.5.)

Upon successful completion, this routine returns a pointer to a valid RPC client. Otherwise, it returns the value NULL.

SVCXPRT *
svctcp_create (sock, sendsz, recvsz)
int sock;
u_int sendsz, rcvcsz;

The **svctcp_create** subroutine creates a RPC service transport based on TCP/IP and returns a pointer to it.

The **sock** parameter specifies the socket associated with the transport. If the **sock** parameter is **RPC_ANYSOCK**, the routine creates a new socket. The transport's socket number is set to **xprt->xp_sock**. If the socket is not bound to a local TCP/IP port, this routine binds it to an arbitrary port. Its port number is set to **xprt->xp_port**.

Since TCP/IP remote procedure calls use buffered I/O, users can set the size of the send and receive buffers with the **sendsz** and **recvsz** parameters. If the size of either buffer is set to 0, the routine picks suitable default values.

On successful completion, this routine returns a valid RPC service transport. If it fails, it returns the value NULL.

SVCXPRT *
svcudp_create (sock)
int sock;

The **svcudp_create** subroutine creates an RPC service transport based on a UDP/IP and returns a pointer to it.

The **sock** parameter specifies the socket associated with the transport. If the **sock** parameter is **RPC_ANYSOCK**, the routine creates a new socket. The transport's socket number is set to **xprt->xp_sock**. If the socket is not bound to a local UDP/IP port, this routine binds it to an arbitrary port. Its port number is set to **xprt->xp_port**.

On successful completion, this routine returns a valid RPC service transport. If it fails, it returns the value NULL.

Note: Use this transport for procedures that take up to 8K bytes of encoded arguments or results only.

xdr_accepted_reply (xdrs, ar)
XDR *xdrs;
struct accepted_reply *ar;

The **xdr_accepted_reply** routine describes RPC messages externally. Use this routine to generate message replies similar to RPC's message replies without using the RPC program.

The **xdrs** parameter points to the XDR stream handle. The **ar** parameter points to the address of the structure that contains the reply.

xdr_authunix_parms (xdrs, app)
XDR *xdrs;
struct authunix_parms *app;

The **xdr_accepted_reply** routine describes credentials externally. Use this routine to generate credentials without using the RPC authentication program.

The **xdrs** parameter points to the XDR stream handle. The **app** parameter points to the structure that contains the authentication credentials.

void
xdr_callhdr (xdrs, chdr)
XDR *xdrs;
struct rpc_msg *chdr;

The **xdr_callhdr** routine describes RPC call headers externally. Use this routine to generate call headers that are similar to RPC's call headers without using the RPC program.

The **xdrs** parameter points to the XDR stream handle. The **chdr** parameter points to the structure that contains the header for the call message.

xdr_callmsg (xdrs, cmsg)
XDR *xdrs;
struct rpc_msg *cmsg;

The **xdr_callmsg** routine describes RPC messages externally. Use this routine to generate messages that are similar to RPC's messages without using the RPC program.

The **xdrs** parameter points to the XDR stream handle. The **cmsg** parameter points to the structure that contains the text of the call message.

xdr_opaque_auth (xdrs, ap)
XDR *xdrs;
struct opaque_auth *ap;

The **xdr_opaque_auth** routine describes RPC messages externally. Use this routine to generate opaque message data without using the RPC program.

The **xdrs** parameter points to the XDR stream handle. The **ap** parameter points to the structure that contains the message text.

xdr_pmap (xdrs, regs)
XDR *xdrs;
struct pmap *regs;

The **xdr_pmap** routine describes parameters for **portmap** procedures externally. Use this routine to generate **portmap** parameters without using the **portmap** interface.

The **xdrs** parameter points to the XDR stream handle. The **regs** parameter points to the buffer, or register, where the **portmap** daemon stores the information.

xdr_pmaplist (xdrs, rp)
XDR *xdrs;
struct pmaplist **rp;

The **xdr_pmaplist** routine describes a list of port mappings externally. Use this routine to generate the port mappings to RPC ports without using the **portmap** interface.

The **xdrs** parameter points to the XDR stream handle. The **rp** parameter is a pointer to the structure that contains the **portmap** listings.

xdr_rejected_reply (xdrs, rr)
XDR *xdrs;

```
struct rejected_reply *rr;
```

The **xdr_rejected_reply** routine describes RPC message rejection replies externally. Use this routine to generate rejection replies similar to RPC's rejection without using the RPC program.

The **xdrs** parameter points to the XDR stream handle. The **rr** parameter points to the structure that contains the rejected reply.

```
xdr_replymsg (xdrs, rmsg)
XDR *xdrs;
struct rpc_msg *rmsg;
```

The **xdr_replymsg** routine describes RPC message replies externally. Use this routine to generate message replies similar to RPC's replies without using the RPC program.

The **xdrs** parameter points to the XDR stream handle. The **rmsg** parameter points to the structure containing the parameters of the reply message.

```
xprt_register (xprt)
SVCXPRT *xprt;
```

The **xprt_register** routine registers an RPC service transport handle with the RPC program after the transport has been created. The **xprt** parameter points to the newly created RPC service transport handle. This routine modifies the **svc_fds** global variable.

Note: Service implementors do not usually need this routine.

```
void
xprt_unregister (xprt)
SVCXPRT *xprt;
```

The **xprt_unregister** routine removes an RPC service transport handle from the RPC service program before the transport handle can be destroyed. The **xprt** parameter points to the RPC service transport handle to be destroyed. This routine modifies the **svc_fds** global variable.

Note: Service implementors do not usually need this routine.

Related Information

In this book: "Remote Procedure Call Service Routines" in topic 1.2.232 and "XDR (External Data Representation)" in topic 1.2.332.

The Network File System section in *Managing the AIX Operating System*.

1.2.232 Remote Procedure Call Service Routines

Purpose
Supports RPC commands and utilities.

Library
Standard C Library (**libc.a**)

Syntax
#include <rpcsvc/*.h>

Description
The Remote Procedure Call Service Routines are used by the Remote Procedure Call (RPC) commands and utilities that are installed with the Network File System (NFS). NFS allows users to share files located on other network machines by using the RPC interface to handle the remote communications.

The following Remote Procedure Call Service Routines are available to programmers as library routines:

getrpcport Gets RPC port numbers.

havedisk Determines if remote machine has a disk.

rnusers Returns number of users on remote machine.

rstat Gets performance data from remote kernel.

rusers Returns information about users on remote machine.

rwall Writes to specified remote machines.

The following Remote Procedure Call Service Routines are not available to programmers as library routines but can be invoked by their RPC program numbers:

ether Monitors network traffic.

rex Executes remote programs.

spray Scatters data packets in order to check the network.

Note: The Remote Procedure Call Service Routines can be invoked through their RPC program numbers by the **callrpc** system call. For information on the **callrpc** system call, see "Remote Procedure Call (RPC)" in topic 1.2.231.

Subtopics
1.2.232.1 Remote Procedure Call Service Routines Available as Library Routines

1.2.232.1 Remote Procedure Call Service Routines Available as Library Routines

The Remote Procedure Call Service Routines that can be used as library routines are listed in this section.

```
int getrpcport (host, prognum, versnum, protocol)
char *host;
int prognum, versnum, protocol;
```

The **getrpcport** routine contacts the **portmap** and returns a port number for an RPC program running on the machine pointed to by the **host** parameter.

The **prognum** and **versnum** parameters identify the program and version numbers of the program. The **protocol** parameter identifies the data transport protocol the program is using (UDP/IP or TCP/IP).

Upon successful completion, this routine returns a valid port number. If the routine can not contact the **portmap** to get the port number or if the program number is not registered, it returns the value 0. If the program number is registered but not with the version specified, it still returns a valid port number.

```
havedisk (host)
char *host;
```

The **havedisk** routine checks to see if the remote computer pointed to by the **host** parameter has a disk. It returns the value 1 if **host** has a disk, and the value 0 if **host** does not have a disk. The value -1 returns if the call fails.

```
#include <rpcsvc/rusers.h>

rnusers (host)
char *host
```

The **rnusers** routine returns the number of users logged in to the remote machine pointed to by the **host** parameter. This routine returns a value -1 if it cannot determine the number.

```
#include <rpcsvc/rusers.h>

rusers (host, up)
char *host;
struct utmpidlearr *up;
```

The **rusers** routine returns status information about users logged in to the remote machine pointed to by the **host** parameter. It returns the information to the **utmpidlearr** structure pointed to by the **up** parameter. The **up** parameter should be initialized to 0 before **rusers** is called. Upon successful completion, the routine returns the value 0.

```
#include <rpc/rpc.h>
#include <rpcsvc/rstat.h>

rstat (host, statp)
char *host;
struct statstime *statp;
```

The **rstat** routine returns performance statistics about users logged in to a remote machine pointed to by the **host** parameter. The information is returned to the **statstime** structure pointed to by the **statp** pointer. Upon successful completion, **rstat** returns the value 0.

```
#include <rpcsvc/rwall.h>

rwall (host, msg);
char *host;
char *msg;
```

The **rwall** routine sends the **msg** parameter to all users logged in to the remote machine pointed to by the **host** parameter. The **msg** parameter points to the text of the message to be printed. Upon successfully writing the message to all users, the **rwall** routine returns the value 0.

Related Information

In this book: "Remote Procedure Call (RPC)" in topic 1.2.231.

1.2.233 rename

Purpose
Renames a directory or a file within a file system.

Syntax

```
int rename (frompath, topath)
char *frompath, *topath;
```

Description
The **rename** system call renames a directory or a file within a file system. The **frompath** and **topath** parameters must both be either files or directories.

For **rename** to execute successfully, the calling process must have write permission to the parent directories of both **frompath** and **topath**, if it already exists.

If *topath* exists, and the parent directory of *topath* has the *sticky* attribute bit set, the calling process must have an effective user ID equal to:

the owner ID of *topath*, or to

the owner ID of the parent directory of *topath*.

The file or directory named by **frompath** cannot contain the file or directory named by **topath**. If **topath** is an existing file or empty directory, it is replaced by **frompath**. If **topath** is a nonempty directory, **rename** exits with an error.

If the file named by **frompath** is a symbolic link, the symbolic link is renamed. If the file named by **topath** is an existing symbolic link, the symbolic link is destroyed.

Return Value
Upon successful completion, the **rename** system call returns a value of 0. If the **rename** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **rename** system call fails and the file or directory name remains unchanged if one or more of the following are true:

ENOTDIR A component of either path prefix is not a directory or **frompath** names a directory and **topath** names a nondirectory.

EISDIR The **topath** parameter names a directory and the **frompath** parameter names a nondirectory.

ENAMETOOLONG
A component of either the **frompath** parameter or the **topath** parameter exceeded **NAME_MAX** characters or the entire parameter exceeded **PATH_MAX** characters.

ENOENT A component of either path does not exist or the file named by **frompath** does not exist.

EACCES Creating the requested link or removing the old link requires

writing in a directory with a mode that denies write permission.

EACCES Search permission is denied on a component of either **frompath** or **topath**.

EPERM The file named by the *topath* parameter is in a directory with the *sticky* attribute bit set, and the effective user ID of the calling process is not equal to the owner of the file or of the parent directory.

EXDEV The link named by **topath** and the file named by **frompath** are on different file systems.

EROFS The named file resides on a read-only file system.

EFAULT Either **frompath** or **topath** points outside of the process's allocated address space.

EBUSY The **frompath** or **topath** directory is currently in use by the system or by another process.

EINVAL The **frompath** or the **topath** is either the current directory or the parent of the current directory.

EINVAL **frompath** is an ancestor directory of **topath**.

EEXIST The **topath** parameter is an existing nonempty directory.

EINTR A signal was caught during the system call.

ESTALE The process's root or current directory is located in an NFS virtual file system that has been unmounted.

EDQUOT The directory in which the entry for the new link is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

Related Information
In this book: "chmod, fchmod" in topic 1.2.44 and "mkdir" in topic 1.2.168.

The **chmod**, **mkdir**, and **mknod**, and **mvdir** commands in *AIX Operating System Commands Reference*.

1.2.234 resolver: res_mkquery, res_send, res_init, dn_comp, dn_expand, getshor

Purpose
Makes, sends, and interprets name server information.

Library
Internet Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

res_query (name, class, type, answer, anslen)
    char *name;
    int class, type;
    u_char *answer;
    int anslen;

res_search (name, class, type, answer, anslen)
    char *name;
    int class, type;
    u_char *answer;
    int anslen;

res_querydomain (name, domain, class, type, answer, anslen)
    char *name; *domain;
    int class, type;
    u_char *answer;
    int anslen;

int res_mkquery(op, dname, class, type, data, datalen, resource buf, buflen)
int op;
char *dname;
int class, type;
char *data;
int datalen;
struct rrec *resource;
char *buf;
int buflen;

int res_send (msgp, msglen, answer, anslen)
char *msgp;
int msglen;
char *answer;
int anslen;

res_init()
int dn_comp(exp_dn, comp_dn, length, dnptrs, lastdnptr)
char *exp_dn, *comp_dn;
int length;
char **dnptrs, **lastdnptr;

int dn_expand(msgp, eomorig, comp_dn, exp_dn, length)
char *msgp, *eomorig
char *comp_dn, *exp_dn;
int  length;
```

```
dn_expand (msg, eomorig, comp_dn, exp_dn, length)
    u_char *msg, *eomorig *comp_dn, *exp_dn;
    int length;

dn_find (exp_dn, msg, dnptrs, lastdnptr)
char *exp_dn, *msg;
char **dnptrs **lastdnptr;

unsigned short getshort (msgp)
char *msgp;

unsigned long getlong (msgp)
char *msgp;

putshort (short, msgp)
unsigned short short;
char *msgp;

putlong (long, msgp)
unsigned long long;
char *msgp;
```

Description

The **res_query, res_mkquery, res_search, res_send, res_init, res_querydomain, dn_comp, dn_expand, getshort, getlong, putshort,** and **putlong** subroutines are used to make, send, and interpret packets for name servers in the Internet domain. Together these subroutines form the ***resolver***, a set of functions that resolves domain names.

Global information that is used by these resolver subroutines is kept in the **_res** structure. This structure is defined in the **resolv.h** header file, and it contains the following members:

```
       int       retrans;
       int       retry;
       long      options;
       int       nscount;
       struct    sockaddr_in nsaddr_list[MAXNS];
       ushort    id;
       char      defdname[MAXDNAME]
  #define nsaddr   nsaddr_list[0]
```

The options field of the **_res** structure is constructed by logically ORing the following values:

RES_INIT Indicates whether the initial name server and default domain name have been initialized (that is, whether **res_init** has been called).

RES_DEBUG Prints debugging messages.

RES_USEVC Uses TCP instead of UDP connections for queries.

RES_STAYOPEN Used with **RES_USEVC** to keep the TCP connection open between queries. While UDP is the mode normally used, TCP mode and this option are useful for programs that regularly do many queries.

RES_RECURSE Sets the recursion desired bit in queries. This is the

default. (**res_send** does not do iterative queries and expects the name server to handle recursion.)

RES_DEFNAMES Appends the default domain name to single label queries. This is the default.

The **res_mkquery** subroutine makes a standard query message and places it in the location pointed to by the **buf** parameter, which has a length that is specified by the **buflen** parameter. The **op** parameter is usually **QUERY** but can be set to any of the query types defined in the **arpa/nameser.h** header file, as listed below:

QUERY Standard query

IQUERY Inverse query

CQUERYM Completion query (multiple)

CQUERYU Completion query (unique)

The **dname** parameter points to the name of the domain. If the value pointed to by **dname** is a single label and the **RES_DEFNAMES** bit is set, as it is by default, **dname** is appended with the current domain name. The current domain name is defined by the name server in use or in the **/etc/resolv.conf** file.

The **class** parameter has one of the following values:

C_IN Specifies the ARPA Internet

C_CHAOS Specifies the chaos network at MIT

The **type** parameter has a value taken from the following list:

T_A Host address

T_NS Authoritative server

T_MD Mail destination

T_MF Mail forwarder

T_CNAME Cannonical name

T_SOA Start of authority zone

T_MB Mailbox domain name

T_MG Mail group member

T_MR Mail rename name

T_NULL NULL resource record

T_WKS Well-known service

T_PTR Domain name pointer

T_HINFO Host information

T_MINFO Mailbox information

T_MX Mail routing information

T_UINFO User (**finger**) information

T_UID User ID

T_GID Group ID

The **data** parameter is a pointer to the data to be sent to the name server as a search key, and the **datalen** parameter defines the size of that data.

The **resource** parameter is a pointer to resource records.

The **res_query** subroutine formulates a normal query, sends, and awaits answer. The returned answer is placed in the supplied buffer **answer**. It performs a preliminary check of **answer** and if no error is indicated and the answer count is nonzero, **res_query** returns the size of the response on success only; on error, it returns a value of -1. The error number is left in **h_errno**. The caller must parse answer and determine whether it answers the question.

The **res_search** subroutine formulates a normal query, sends, and retrieves the answer in the supplied buffer. **res_search** returns the size of the response on success and a value of -1 on error. If enabled, **res_search** implements search rules until **answer** or unrecoverable failure is detected. The error number is left in **h_errno**. This subroutine is only useful for queries in the same name hierarchy as the local host.

The **res_querydomain** subroutine performs a call on **res_query** on the concatenation of name and domain, removing a trailing dot from name if the domain is **NULL**.

The **res_send** subroutine sends a query to name servers, calling the **res_init** subroutine if **RES_INIT** is not set. This subroutine sends the query to the local name server and handles timeouts and retries.

The **res_init** subroutine reads the **/etc/resolv.conf** file for the default domain name and Internet address of the initial hosts running the name server. If this line does not exist, the **res_init** subroutine tries the host from which it was called.

The **dn_comp** subroutine compresses the domain name pointed to by the **exp_dn** parameter and stores it in the area pointed to by the **comp_dn** parameter. The **length** parameter is the size of the array pointed to by the **comp_dn** parameter. The **dnptrs** parameter is a list of pointers to previously compressed names in the current message. The first pointer points to the beginning of the message and the list ends with NULL. The **lastdnptr** parameter is a pointer to the end of the array pointed to by the **dnptrs** parameter.

A side effect of this subroutine is to update the list of pointers for labels that the **dn_comp** subroutine inserts into the message as the name is compressed. No names are compressed if the value of **dnptrs** is NULL. If the **lastdnptr** parameter is NULL, the list of pointers is not updated.

The **dn_expand** subroutine expands the compressed domain name pointed to by the **comp_dn** parameter to a full domain name, converting the expanded names to uppercase. The **msgp** parameter is a pointer to the beginning of the

message. The **exp_dn** parameter is a pointer to a buffer, of the size specified by the **length** parameter, that holds the result. The **emorig** parameter points to the end of the original message, which contains the compressed domain name.

The **dn_skipname** subroutine skips over a compressed domain name.

The **dn_find** subroutine searches for an expanded domain name from a list of previously compressed names and returns the offset from *msg*, if found.

The **getshort** and **getlong** subroutines get quantities from the byte stream or arbitrary byte boundaries. The **putshort** and **putlong** subroutines put quantities into the byte stream or arbitrary byte boundaries. The **msgp** parameter for all of these subroutines represents a pointer into the byte stream.

Return Value

The **res_mkquery** subroutine returns the size of the query on success. A value of -1 is returned if the subroutine fails because the query is larger than the value of the **buflen** parameter. The **res_send** subroutine returns the length of the message if it succeeds and a value of -1 if it fails. The **dn_comp** subroutine returns the size of the compressed domain name if it succeeds. A value of -1 is returned if the subroutine fails. The **dn_expand** subroutine returns the size of the expanded domain name on success. A value of -1 is returned if **dn_expand** fails. The **getshort** and **getlong** subroutines return a **short** and a **long** value, respectively.

File

/etc/resolv.conf Contains name server and domain name information.

Related Information

See "resolver" in *AIX TCP/IP User's Guide*, Chapter 2.

1.2.235 rexec

Purpose
Allows command execution on a remote Internet host.

Library
Internet Library (**libc.a**)

Syntax

int rexec (host, port, user, passwd, command, errfdp)

char **host;
int port;
char *user, *passwd, *command;
int *errfdp

Description

The **rexec** subroutine allows the calling process to execute commands on a remote host which usually is outside your TCF cluster.

The **rexec** subroutine uses the **gethostbyname** subroutine to find the host specified by **host**. ***host** is updated to point to the standard name of the host found by **gethostbyname**. If the host does not exist, the **rexec** subroutine fails and returns -1.

The **port** parameter specifies the well-known DARPA Internet port to use for the connection. A pointer to the structure that contains the necessary port can be obtained by issuing the following call:

```
getservbyname("exec", "tcp")
```

The protocol for the connection is described in detail in the discussion of **rexecd** in *AIX TCP/IP User's Guide*.

The **user** and **passwd** parameters point to a user ID and password valid at the host. If these parameters are not supplied, the **rexec** subroutine takes the following actions until finding a user ID and password to send to the remote host:

1. Searches the current environment for the user ID and password on the remote host.

2. Searches the user's home directory for a file called **.netrc** that contains a user ID and password.

3. Prompts the user for a user ID and password.

The **command** parameter points to the name of the command to be executed at the remote host.

If the connection succeeds, a socket in the Internet domain of type **SOCK_STREAM** is returned to the calling process and is given to the remote command as standard input and standard output.

If **errfdp** is not 0, an auxiliary channel to a control process is set up, and a descriptor for it is placed in ***errfdp**. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group

of the command. This diagnostic information does not include remote authorization failure, since this connection is set up after authorization has been verified.

If **errfdp** is 0, then the standard error of the remote command is the same as standard output, and no provision is made for sending arbitrary signals to the remote process. In this case, however, it may be possible to send out-of-band data to the remote command.

Return Value

The **rexec** subroutine fails and a value of -1 is returned if the specified host name does not exist.

Related Information

In this book: "rcmd, rresvport, ruserok" in topic 1.2.223.

The discussion of **rexecd** and **/etc/hosts** in *AIX TCP/IP User's Guide*.

1.2.236 rexec: rexecl, rexecv, rexecle, rexecve, rexeclp, rexecvp

Purpose

Executes a file on a given site.

Syntax

```
#include <sys/types.h>

rexecl(path, arg0, arg1, ..., argn, (char *)0, site_number)
char *path, *arg0, *arg1, ..., *argn;
siteno_t site_number;

rexecv(path, argv, site_number)
char *path, *argv[ ];
siteno_t site_number;

rexecle(path, arg0, arg1, ..., argn, (char *)0, envp,  site_number)
char *path, *arg0, *arg1, ..., *argn, *envp[ ];
siteno_t site_number;

rexecve(path, argv, envp, site_number)
char *path, *argv[ ], *envp[ ];
siteno_t site_number;

rexeclp(file, arg0, arg1, ..., argn, (char *)0, site_number)
char *file, *arg0, *arg1, ..., *argn;
siteno_t site_number;

rexecvp(file, argv, site_number)
char *file, *argv[ ];
siteno_t site_number;
```

Description

The **rexec** family of system calls is only available with the Transparent Computing Facility. The **rexec** family of system calls is just like the **exec** family, except that it allows a site to be specified on which the new process file will execute. If **site_number** is 0, **rexec** is exactly the same as **exec**.

The new process executes on the site specified by **site_number**. If **site_number** is 0, the site is determined by the machine type on which the new process file must run and by the site path (see "getspath, setspath" in topic 1.2.122).

If the name of the new process file refers to a hidden directory and **site_number** is nonzero, **rexec** selects the new process file by using a site path which consists only of the machine type corresponding to **site_number**. The new process's site path is not affected.

See "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71 for details concerning the arguments to the various forms of **rexec** which are analogous to those of **exec**.

Note: The **rexec** family of system calls may not execute a new program at another site if the calling process has too many (85 or more) child processes. A process may not use the **rexec** family of system calls to execute a new program at another site if the process has a file open which is marked as being in error (for instance, the storage

site is not on the network) or if the process has a character device open (other than a terminal or the null device).

The **rexec** family of system calls may not execute a new program at another site if the calling process has made use of shared memory.

Return Value
If **rexec** returns to the calling process, an error has occurred; the return value is -1 and **errno** is set to indicate the error.

Error Conditions
The **rexec** system call fails for any of the reasons listed for **exec** plus:

EBADST **site_number** is out of range.

Files

/bin/sh Invoked if the shell program is found by **rexeclp** or **rexecvp**.

Related Information
In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, " fork, vfork" in topic 1.2.83, "getspath, setspath" in topic 1.2.122, "getxperm, setxperm" in topic 1.2.128, "migrate" in topic 1.2.167, "rfork" in topic 1.2.237, " run: runl, runv, runle, runve, runlp, runvp" in topic 1.2.239, "a.out" in topic 2.3.2, and "environment" in topic 2.4.6.

1.2.237 rfork

Purpose
Creates a new process on another site.

Syntax

```
#include <sys/types.h>

pid_t rfork (site_number)
siteno_t site_number;
```

Description
The **rfork** system call is only available with the Transparent Computing Facility. The **rfork** system call is comparable to the **fork** system call, except that it allows a site to be specified on which the child process is created. If **site_number** is 0, the child is created locally.

The **rfork** system call causes the creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process:

Environmen

Close-on-exec flag (see "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71)

Signal handling settings (that is, **SIG_DFL**, **SIG_IGN**, function address)

Signal mechanism new/old status (see "sigaction, sigvec, signal" in topic 1.2.263)

Set-user-ID and set-group-ID mode bits (see "setxuid" in topic 1.2.256)

Profiling on/off statu

Nice value (see "getpriority, setpriority, nice" in topic 1.2.111)

All attached shared memory segments (see "Semaphores, Message Queues, and Shared Memory Segments" in topic 1.2.2.10)

Note: If there are any shared memory segments, **rfork** fails unless the child process's site is the same as the parent's site.

Process group I

Session I

TTY group ID (see "exit, _exit" in topic 1.2.73 and "sigaction, sigvec, signal" in topic 1.2.263)

Current working director

Root director

<LOCAL> alias path name (see "getlocal, setlocal" in topic 1.2.102)

File mode creation mask (see "umask" in topic 1.2.314)

System resource limits (se "getrlimit, setrlimit, vlimit" in topic 1.2.115 and "ulimit" in topic 1.2.313)

Site path (see "getspath, setspath" in topic 1.2.122)

Execution site permissions (see "getxperm, setxperm" in topic 1.2.128).

The child process differs from the parent process in the following ways:

The child process has a unique process ID

The child process has a different parent process ID (that is, th process ID of the parent process).

The child process has its own copy of the parent's file descriptors Each of the child process's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

The trace flag is cleared (see "ptrace" in topic 1.2.212).

All **semadj** values are cleared (see "semop" in topic 1.2.245).

Process locks, text locks, data locks, and file locks are no inherited by the child process (see "plock" in topic 1.2.205 and "fcntl, flock, lockf" in topic 1.2.78).

The child process's **utime**, **stime**, **cutime**, and **cstime** are set to 0.

The time left until an alarm clock signal is reset to 0

Note: Processes may not fork to another site if the destination site has a different cpu type or if they have too many (85 or more) child processes. Processes may not use the **rfork** system call to create a new process on another site if they have made use of shared memory, semaphores, or message operations (see "Semaphores, Message Queues, and Shared Memory Segments" in topic 1.2.2.10). Processes may not use the **rfork** system call to create a new process on another site if they have a file open which is marked as being in error (for example, if the storage site is not on the network) or if they have a character device open (other than a terminal or the null device).

Return Value
Upon successful completion, **rfork** returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and **errno** is set to indicate the error.

Error Conditions
The **rfork** system call fails and no child processes are created if one or more of the following are true:

EAGAIN The user is the superuser, and the system-imposed limit on the total number of processes under execution on the remote site would be exceeded.

EAGAIN The user is the not superuser, and the system-imposed limit on the total number of processes under execution by a single user on the remote site would be exceeded.

EBADST **site_number** is out of range or the destination site is not the same CPU type as the current site.

EPERM Execute permission is not granted for **site_number**.

ESITEDN1 The operation failed because a required site is unavailable.

ESITEDN2 The operation was terminated because a site failed.

ENOMEM There is not enough space left for this process on the new site.

ETABLE On either the parent's site, the system's PID-site table, which is used to keep track of remote processes and process groups, is full.

ELOCALONLY
The process may not remote fork because it is using semaphores, message queues, and shared memory (see "Semaphores, Message Queues, and Shared Memory Segments" in topic 1.2.2.10) or it has too many child processes.

ENLDEV The process may not execute on the designated site because one of its open file descriptors is for a local-only object such as a socket or a non-**tty** character special file.

ENOSTORE The current load module cannot be located from site *sitenumber* because it has been deleted or superceded by a new version.

Related Information
In this book: "semop" in topic 1.2.245, "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, " fork, vfork" in topic 1.2.83, "getlocal, setlocal" in topic 1.2.102, "getpriority, setpriority, nice" in topic 1.2.111, "getrlimit, setrlimit, vlimit" in topic 1.2.115, "getspath, setspath" in topic 1.2.122, "getxperm, setxperm" in topic 1.2.128, "migrate" in topic 1.2.167, "plock" in topic 1.2.205, "ptrace" in topic 1.2.212, " rexec: rexecl, rexecv, rexecle, rexecve, rexeclp, rexecvp" in topic 1.2.236, " run: runl, runv, runle, runve, runlp, runvp" in topic 1.2.239, "Semaphores, Message Queues, and Shared Memory Segments" in topic 1.2.2.10, "sigaction, sigvec, signal" in topic 1.2.263, "times" in topic 1.2.304, "ulimit" in topic 1.2.313, "umask" in topic 1.2.314, "wait, waitpid" in topic 1.2.325, and "wait3" in topic 1.2.326.

1.2.238 rmdir

Purpose
Removes a directory file.

Syntax

rmdir (path)
char *path;

Description
The **rmdir** system call removes the directory specified by the **path** parameter. The directory you specify must be empty and you must have write access to it.

If the parent directory of *path* has the *sticky* attribute bit set, the calling process must have an effective user ID equal to:

the owner ID of *path*, or to

the owner ID of the parent directory of *path*.

Return Value
Upon successful completion, the **rmdir** system call returns a value of 0. If the **rmdir** system call fails, a value of -1 is returned, and **errno** is set to indicate the error. It is an error to apply **rmdir** to a symbolic link.

Error Conditions
The **rmdir** system call fails and the directory is not deleted if one or more of the following are true:

EBUSY The directory is in use as either the mount point for a file system or the current directory of the process that issued the **rmdir**.

EEXIST The directory is not empty.

ENOTDIR A component of the path is not a directory.

ENOENT The named file does not exist.

EACCES A component of the path denies search permission or write permission is denied on the directory containing the link to be removed.

EROFS The named file resides on a read-only file system.

EFAULT **path** points outside of the process's allocated address space.

ESTALE The process's root or current directory is located in an NFS virtual file system that has been unmounted.

ENAMETOOLONG
A component of the **path** parameter exceeded **NAME_MAX** characters or the entire **path** parameter exceeded **PATH_MAX** characters.

EISDIR A hidden directory was named, components within hidden directories must be explicitly named.

ENOENT A symbolic link was named in the path prefix, but the file to which it refers does not exist.

ELOOP A loop of symbolic links was detected.

ENFILE The system inode table is full.

EPERM The file named by the *path* parameter is in a directory with the *sticky* attribute bit set, and the effective user ID of the calling process is not equal to the owner of the file or of the parent directory.

If the Transparent Computing Facility is installed on your system, **rmdir** can also fail if one or more of the following are true:

ELOOP An **rmdir** was attempted on a symbolic link in a system-type replicated file system.

ESITEDN1 **path** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE **path** is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of **path** is replicated but not stored on any site which is currently up.

EROFS Write access is requested for a file on a replicated file system in which the primary copy is unavailable.

EINTR A signal was caught during the system call.

Related Information
In this book: "chmod, fchmod" in topic 1.2.44, "mkdir" in topic 1.2.168, "mknod, mknodx, mkfifo" in topic 1.2.169, "rename" in topic 1.2.233, and "umask" in topic 1.2.314.

1.2.239 run: runl, runv, runle, runve, runlp, runvp

Purpose
Runs a file.

Syntax

```
#include <sys/types.h>

pid_t runl(path, arg0, arg1, ..., argn, (char *)0, site_number, fdmapsize, fdm
char *path, *arg0, *arg1, ..., *argn, *fdmap;
siteno_t site_number;
int fdmapsize;

pid_t runv(path, argv, site_number, fdmapsize, fdmap)
char *path, *argv[ ], *fdmap;
siteno_t site_number;
int fdmapsize;

pid_t runle(path, arg0, arg1, ..., argn, (char *)0, envp, site_number, fdmapsi
char *path, *arg0, *arg1, ..., *argn, *envp[ ], *fdmap;
siteno_t site_number;
int fdmapsize;

pid_t runve(path, argv, envp, site_number, fdmapsize, fdmap)
char *path, *argv[ ], *envp[ ], *fdmap;
siteno_t site_number;
int fdmapsize;

pid_t runlp(file, arg0, arg1, ..., argn, (char *)0, site_number, fdmapsize, fd
char *file, *arg0, *arg1, ..., *argn, *fdmap;
siteno_t site_number;
int fdmapsize;

pid_t runvp(file, argv, site_number, fdmapsize, fdmap)
char *file, *argv[ ], *fdmap;
siteno_t site_number;
int fdmapsize;
```

Description
The **run** family of system calls is only available with the Transparent Computing Facility. These calls in all their forms create a new process, overlay it with the named file and transfer to the entry point for that file in the new core image. They are essentially combinations of the **fork** and **exec** system calls; however, much of the internal overhead of **fork** is avoided, providing a more efficient implementation. Also, they verify that the **exec** system call is going to succeed before the **fork** system call is attempted.

The new process is created on the site specified by **site_number**. If **site_number** is 0, the site is chosen using the process's site path (see "getspath, setspath" in topic 1.2.122).

If **fdmap** is nonzero, it specifies from where each of the child process's file descriptors will be copied. The value stored at **fdmap[i]** specifies which of the parent's file descriptors (if any) will become the child process's **i**-th descriptor. The child process's **i**-th descriptor are be closed if any of the following are true:

fdmap[i] is not a valid file descriptor number (-1 is used by convention to get a closed descriptor in the child process).

fdmap[i] is not an open file descriptor in the parent.

fdmap[i] is equal to i, and the **close-on-exec** flag is set in the parent for this file descriptor.

Under no circumstances are the parent's files be affected by this mapping. If **fdmap** is 0, the child process's files are be copied from the parent's as in a **fork** call.

The argument **fdmapsize** specifies the number of files descriptors contained in the **fdmap**. All file descriptors greater than or equal to **fdmapsize** are copied from the parent process as is done in a **fork** call.

See "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71 for details concerning the arguments to the various forms of **run** which are analogous to those of **exec**.

Note: Processes may not run on another site if the parent process has too many (85 or more) child processes. Processes may not run on another site if the parent process has a file open which is marked as being in error (for example, if the storage site is not on the network) or if it has a character device open (other than a terminal or the null device).

Return Value
Upon successful completion, the **run** family returns the process ID of the child process to the parent process. It does not return in the child process; the child begins execution of the new file. If an error occurs, a value of (PID_t) -1 is returned to the parent process, no child process is created (except, possibly, in the case of **ESITEDN2**), and **errno** is set to indicate the error.

Error Conditions
The **run** system calls may fail for any of the reasons listed for the **fork** and **rexecvp** system calls plus:

EFAULT **fdmap** points to an illegal address.

EINTR A signal was caught during the system call.

File

/bin/sh Invoked if a shell program is found by **runlp** or **runvp**.

Related Information
In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, " fork, vfork" in topic 1.2.83, "getspath, setspath" in topic 1.2.122, "getxperm, setxperm" in topic 1.2.128, "migrate" in topic 1.2.167, " rexec: rexecl, rexecv, rexecle, rexecve, rexeclp, rexecvp" in topic 1.2.236, "rfork" in topic 1.2.237, "a.out" in topic 2.3.2, and "environment" in topic 2.4.6.

1.2.240 scandir

Purpose
Scans a directory.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <dirent.h>

int scandir (dirname, namelist, select, compar)
char *dirname;
struct dirent * (*namelist [ ]);
int  (*select) ( );
int  (*compar) ( );
```

Description
The **scandir** subroutine reads the directory pointed to by **dirname**, then uses the **malloc** subroutine to create an array of pointers to directory entries. The **scandir** subroutine returns the number of entries in the array and, through the **namelist** parameter, a pointer to the array.

The **select** parameter points to a user-supplied subroutine that is called by **scandir** to select which entries to include in the array. The selection routine is passed a pointer to a directory entry and should return a nonzero value for a directory entry that is included in the array. If **select** is NULL, all directory entries are included.

The **compar** parameter also points to a user-supplied subroutine. This routine is passed to **qsort** to sort the completed array. If **compar** is NULL, the array is not sorted. The **alphasort** subroutine provides comparison functions for sorting alphabetically and can be specified by the **compar** parameter. (See "alphasort" in topic 1.2.15.)

The memory allocated to the array can be deallocated by freeing each pointer in the array and the array itself, with the **free** subroutine. (For more information on **free**, see "malloc, free, realloc, calloc, valloc, alloca, mallopt, mallinfo" in topic 1.2.162.)

Return Value
The **scandir** subroutine returns the value -1 if the directory cannot be opened for reading, or if the **malloc** subroutine cannot allocate enough memory to hold all the data structures. If successful, the **scandir** subroutine returns the number of entries found.

Related Information
In this book: "alphasort" in topic 1.2.15, "directory: opendir, readdir, telldir, seekdir, rewinddir, closedir" in topic 1.2.60, "malloc, free, realloc, calloc, valloc, alloca, mallopt, mallinfo" in topic 1.2.162, "qsort" in topic 1.2.217, and "dir" in topic 2.3.16.

1.2.241 scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf, wsscanf

Purpose
Converts formatted input.

Library
Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>


int scanf (fmt [, ptr, ... ]int NLscanf (fmt [, ptr, ... ])
char *fmt;                   char *fmt;

int fscanf (stream, fmt [, pint NLfscanf (stream, fmt [, ptr,
FILE *stream;                FILE *stream;
char *fmt;                   char *fmt;

int sscanf (s, fmt [, ptr, .int]NLsscanf (s, fmt [, ptr, ... ]
char *s, *fmt;               char *s, *fmt;
int wsscanf (wcs, fmt [, ptr, ...])
wchar_t *wcs;
char *fmt;
```

Description
The **scanf**, **fscanf**, and **sscanf** subroutines read character data, interpret it according to a format, and store the converted results into specified memory locations. The **NLscanf**, **NLfscanf**, and **NLsscanf** subroutines parallel their corresponding functions, providing conversion types to handle **NLchar**s as well as **char**s. The **wsscanf** subroutine is equivalent to **sscanf**, except that the argument **wcs** specifies a wide character string from which the input is obtained, rather than a string. Reaching the end of the wide character string is equivalent to reaching the end of string for **sscanf**. If copying takes place between objects that overlap, the behavior is undefined. These subroutines read their input from the following sources:

scanf, **NLscanf**	Reads from standard input (**stdin**).
fscanf, **NLfscanf**	Reads from **stream**.
sscanf, **NLsscanf**	Reads from the character string **s**.
wsscanf	Reads from the wide character string **s**.

The **fmt** parameter contains conversion specifications used to interpret the input. The **ptr** parameters specify where to store the interpreted data.

The **fmt** parameter can contain the following:

White space characters (blanks, tabs, new-lines, or form-feeds) which except in two cases described following, reads the input up to the next nonwhite space character. Unless there is a match in the control string, trailing white space (including a new-line character) is not read.

Any character except % (percent), which must match the next one or more characters of the input stream.

A conversion specification that directs the conversion of the nex

input field. It consists of the following:

- The character **%** (percent)
- An optional assignment suppression character, ***** (asterisk)
- An optional numeric maximum field width
- An optional character that sets the size of the receiving variable as for some flags, as follows:
  - **l** Signed long integer rather than an **int** when preceding the **d**, **u**, **o** or **x** conversion codes. A double rather than a float, when preceding the **e**, **f** or **g** conversion codes.
  - **h** Signed short integer (half **int**) rather than an **int** when preceding the **d**, **u**, **o** or **x** conversion codes.
- A conversion code.

Each conversion specification in the **fmt** parameter has a **%** (percent sign) or the character sequence **%digit$**, which introduces the conversion specification.

Conversions can be applied to the **n**th argument in the argument list, rather than to the next unused argument. In this case, the conversion character **%** is replaced by the sequence **%digit$**, where **digit** is a decimal integer **n** in the range of **[1, {NL_ARGMAX}]**, giving the position of the argument inthe argument list. With this feature, format strings can be defined to assure that arguments are selected in an order appropriate for the specified language.

%[*][**width**][**size]convcode**

The results from the conversion are placed in ***ptr** unless you specify assignment suppression with *****. Assignment suppression provides a way to describe an input field that is to be skipped. The input field is a string of nonwhite-space characters. It extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion code indicates how to interpret the input field. The corresponding **ptr** must usually be of a restricted type. You should not specify **ptr** for a suppressed field. You can use the following conversion codes:

% Accepts a single **%** input at this point; no assignment is done.

d Accepts a decimal integer; **ptr** should be an integer pointer.

u Accepts an unsigned decimal integer; **ptr** should be an unsigned integer pointer.

o Accepts an octal integer; **ptr** should be an integer pointer.

x Accepts a hexadecimal integer; **ptr** should be an integer pointer.

e, **f**, **g** Accepts a floating-point number. The next field is converted accordingly and stored through the corresponding parameter, which should be a pointer to a **float**. The input format for floating-point numbers is a string of digits, with some optional characteristics:

It can be a signed value.
It can be an exponential value, containing a decimal point followed by an exponent field, which consists of an **E** or an

e followed by an (optionally signed) integer. It can be one of the special values **INF**, **QNaN**, or **SNaN**, which is translated into the ANSI/IEEE value for infinity, quiet NaN, or signalling NaN, respectively.

s Accepts a string of **char**s. The **ptr** parameter should be a character pointer that points to an array of characters large enough to accept the string and ending with **'\0'**. The **'\0'** is added automatically. The input field ends with a white space character. A string of **char**s is output.

S (Used by the **NLscanf**, **NLfscanf**, and **NLsscanf** subroutines only.) Accepts an **NLchar** string. The **ptr** parameter should point to an array of characters large enough to accept the string and end with **'\0'**. The **'\0'** is added automatically. The input field ends with a white space character. A string of **NLchar**s is output.

N (Used by the **NLscanf**, **NLfscanf**, and **NLsscanf** subroutines only.) Accepts an ASCII string, possibly containing extended character information in the form of escape sequences used by the **NLescstr** and **NLunescstr** subroutines. (See "display symbols" in topic 2.4.4 for a list of these escape sequences.) The output is in the form of **NLchar**s.

c A character is expected. The **ptr** parameter should be a character pointer. The normal skip over white space is suppressed. Use **%1s** to read the next nonwhite-space character. If a field width is given, **ptr** should refer to a character array; the indicated number of characters is read.

wc Matches a sequence of multibyte characters of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument is a pointer to the initial wide character of an array large enough to accept the sequence resulting from the conversion. This conversion is the same as that performed by the **mbstowcs** routine.

ws Matches a sequence of non-white space multibyte characters. The corresponding argument is a pointer to the initial wide character of an array large enough to accept the sequence resulting from the conversion and a terminating NULL wide character, which is added automatically. This conversion is the same as that performed by the **mbstowcs** subroutine.

[**scanset**] Accepts as input the characters included in the ***scanset***. The scanset explicitly defines the characters that are accepted in the string data as those enclosed within square brackets. The normal skip over leading white space is suppressed. A scanset in the form of **[^scanset]** is an ***exclusive scanset***: the ^ (circumflex) serves as a complement operator and the following characters in the scanset are not accepted as input. Conventions used in the construction of the **scanset** follow:

You can represent a range of characters by the construct **first-last**. Thus you can express [0123456789] as [0-9]. The **first** parameter must be lexically less than or equal to **last**, or else the - (dash) stands for itself. The - also stands for itself whenever it is the first or the last character in the **scanset**.

The range of characters, which is locale dependent, is determined by the current locale's collation table.

You can include the] (right bracket), as an element of the **scanset**, if it is the first character of the **scanset**. In this case it is not interpreted as the bracket that closes the scanset. If the scanset is an exclusive scanset, the] is preceded by the ^ (circumflex) to make the] an element of the scanset. The corresponding **ptr** must point to a character array large enough to hold the data field and that ends with **'\0'**. The **'\0'** is added automatically.

A **scanf** or **NLscanf** conversion ends at the end-of-file, the end of the control string, or when an input character conflicts with the control string. If it ends with an input character conflict, the character that conflicts is not read from the input stream.

Note: Unless there is a match in the control string, trailing blanks (including a new-line character) are not read.

The success of literal matches and suppressed assignments is not directly determinable.

Return Value

The **scanf** and **NLscanf** subroutines return the number of successfully matched and assigned input items. This number can be 0 if there was an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, only EOF is returned.

The **wsscanf** subroutine returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, **wsscanf** returns the number of input items assigned, which can be fewer than provided for, or even zero in the event of an early matching failure.

Examples

1. To read several values and assign them to variables:

```
int i;
float x;
char name[50];

scanf ("%d%f%s", &i, &x, name);
```

   with the input line:

```
25 54.32E-1 thompson
```

   This assigns to **i** the value **25**, to **x** the value **5.432**, and to **name** the value **thompson\0**.

2. To perform simple pattern-matching while scanning the input:

```
int i;
float x;
char name[50];

scanf ("%2d%f%*d %[0-9]", &i, &x, name);
```

with the input:

```
  56789 0123 56a72
```

This assigns **56** to **i**, **789.0** to **x**, skips **0123**, and places the string **56\0** in **name**. The next call to **getchar** returns **a**. (See "getc, fgetc, getchar, getw, getwc, fgetwc, getwchar" in topic 1.2.91.)

Related Information

In this book: "getc, fgetc, getchar, getw, getwc, fgetwc, getwchar" in topic 1.2.91, "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf" in topic 1.2.208, " stdio" in topic 1.2.283, "strtod, atof" in topic 1.2.290, "strtol, atol, atoi" in topic 1.2.291, "limits.h" in topic 2.4.11, and "display symbols" in topic 2.4.4.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.242 select

Purpose
Checks the I/O status of multiple file descriptors and message queues.

Syntax

```
#include <sys/select.h>
#include <sys/types.h>
#include <sys/time.h>

int select (nfdsmsgs, readlist, writelist, exceptlist, timeout)
int nfdsmsgs;
struct sellist *readlist, *writelist, *exceptlist;
struct timeval *timeout;

int select (nfds, readfds, writefds, exceptfds, timeout)
int nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;
```

Description

The **select** system call checks the specified file descriptors and message queues to see if they are ready for reading (receiving) or writing (sending), or if they have an exceptional condition pending. There are two user interfaces to **select**. The first one is actually a superset of the second, although that is not obvious from the syntax. The second interface is provided to achieve source code compatibility with software from 4.3BSD systems.

Note: The **select** system call applies only to character devices, pipes, and message queues. Not all character device drivers support it.

See the descriptions of individual character devices in Chapter 5, "Special Files" for information about whether and how specific device drivers support **select**.

The **nfdsmsgs** parameter specifies the number of file descriptors and the number of message queues to check. The low-order 16 bits give the length of a bit mask that specifies which file descriptors to check; the high-order 16 bits give the size of an array that contains message queue identifiers. If either half of the **nfdsmsgs** parameter is equal to 0, it is assumed the corresponding bit mask or array not present.

The **readlist**, **writelist**, and **exceptlist** parameters specify what to check for reading, writing, and exceptions, respectively. Together, they specify the selection criteria. Each of these parameters points to a **sellist** structure, which can specify both file descriptors and message queues. Your program must define the **sellist** structure in the following form:

```
struct sellist
{
   int fdsmask[f];       /* file descriptor bit mask  */
   int msgids[m];        /* message queue identifiers */
};
```

The **fdsmask** array is treated as a bit string in which each bit corresponds to a file descriptor. File descriptor **n** is represented by the bit

(1 << n) in the array element **fdsmask[n / BITS(int)]**. (The **BITS** macro is defined in the **values.h** header file.) Each bit that is set to 1 indicates that the status of the corresponding file descriptor is to be checked. Note that the low-order 16 bits of the **nfdsmsgs** parameter specify the number of **bits** (not elements) in the **fdsmask** array that make up the file descriptor mask. If only part of the last **int** is included in the mask, then the appropriate number of low-order bits are used, and the remaining high-order bits are ignored. If you set the low-order 16 bits of the **nfdsmsgs** parameter to 0, then you must **not** define a **fdsmask** array in the **sellist** structure.

Each **int** of the **msgids** array specifies a message queue identifier whose status is to be checked. Elements with a value of -1 are ignored. The high-order 16 bits of the **nfdsmsgs** parameter specify the number of elements in the **msgids** array. If you set the high-order 16 bits of the **nfdsmsgs** parameter to 0, then you must **not** define a **msgids** array in the **sellist** structure.

If the **timeout** parameter is not a NULL pointer, then it points to a structure that specifies the maximum length of time to wait for at least one of the selection criteria to be met. The **timeval** structure is defined in the **sys/select.h** header file, and it contains the following members:

```
long tv_sec;        Seconds
long tv_usec;       Microseconds
```

The number of microseconds specified in **timeout.tv_usec**, a value from 0 to 999999, is rounded to the nearest second by the AIX Operating System.

If the **timeout** parameter is a NULL pointer, then the **select** system call waits indefinitely, until at least one of the selection criteria is met. If the **timeout** parameter points to a **timeval** structure that contains zeros, then the file and message queue status is polled, and the **select** system call returns immediately.

Note: The arrays specified by **readlist**, **writelist**, and **exceptlist** are the same size because each of these parameters points to the same **sellist** structure type. However, you need not specify the same number of file descriptors or message queues in each. Set the file descriptor bits that are not of interest to 0, and set the extra elements of the **msgids** array to -1.

You can use the **SELLIST** macro defined in the **sys/select.h** header file to define the **sellist** structure. The format of this macro is:

```
SELLIST(f, m)  declarator...;
```

where **f** specifies the size of the **fdsmask** array, **m** specifies the size of the **msgids** array, and each **declarator** is the name of a variable to be declared as having this structure type.

For example, suppose you want to test file descriptors 1, 2, and 35 in addition to five message queues. On the PS/2, which has 32-bit integers, this requires two **int**s for the bit mask. Five **int**s are required to specify the message queue identifiers. The structures can be defined like this:

```
SELLIST(2, 5)  rd, wr, ex;
```

This macro expands to:

```
struct
{
   int  fdsmask[2];
   int  msgids[5];
}  rd, wr, ex;
```

Note that the **SELLIST** macro does not define the structure with a tag (that is, as **struct sellist**).

The **SELLIST** macro cannot be used if you specify either half of the **nfdsmsgs** parameter as 0, indicating that one of the arrays is not present. Trying to use **SELLIST(0,5)**, for example, results in a compiler error caused by defining an array with a dimension of 0. In this case, you must define the structure yourself, including only the desired array.

The arguments to the second interface are for file descriptors only. **nfds** is the number of file descriptors represented in the file descriptor masks. The **readfds**, **writefds**, and **exceptfds** arguments are pointers to masks indicating the file descriptor sets. The **timeout** parameter has the function described above.

The descriptor sets are stored as bit fields in arrays of **int**s. The following macros are provided for manipulating such descriptor sets:

FD_ZERO(&fdset) Initializes a descriptor set **fdset** to the null set.

FD_SET(fd, &fdset) Includes a particular descriptor **fd** in **fdset**.

FD_CLR(fd, &fdset) Removes **fd** from **fdset**.

FD_ISSET(fd, &fdset) Is nonzero if **fd** is a member of **fdset**, 0 otherwise.

The behavior of these macros is undefined if a descriptor value is less than 0 or greater than or equal to **FD_SETSIZE**, which is normally at least equal to the maximum number of descriptors supported by the system.

Return Value

Upon successful completion, the **select** system call returns a value that indicates the total number of file descriptors and message queues that satisfy the selection criteria. The **fdsmask** or **fdset** bit masks are modified so that bits set to 1 indicate file descriptors that meet the criteria. The **msgids** arrays are altered so that message queue identifiers that do not meet the criteria are replaced with a value of -1.

The return value through the first interface is similar to the **nfdsmsgs** parameter in that the low-order 16 bits give the number of file descriptors, and the high-order 16 bits give the number of message queue identifiers. These values indicate the sum total that meet each of the read, write and exception criteria. Therefore, the same file descriptor or message queue may be counted up to three times. The return value through the second interface is the sum total that meets each of the read, write, and exception criteria.

You can use the **NFDS** and **NMSGS** macros to separate out these two values from the return value. If **rc** contains the value returned from the **select** system call, then **NFDS(rc)** is the number of files selected, and **NMSGS(rc)** is the number of message queues selected.

If the **select** system call fails, it returns a value of -1 and sets **errno** to indicate the error. In this case, the contents of the structures pointed to by the **readlist**, **writelist**, and **exceptlist** parameters are unpredictable. If the time limit specified by the **timeout** parameter expires, **select** returns a value of 0.

Error Conditions

The **select** system call fails if one or more of the following is true:

EBADF An invalid file descriptor or message queue identifier is specified.

EINTR A signal was encountered before any of the selected events occurred, or before the time limit expired.

EFAULT The **readlist**, **writelist**, **exceptlist**, or **timeout** parameter points to a location outside of the process's allocated address space.

EINVAL One of the parameters contains an invalid value.

EFAULT The **readfds**, **writefds**, **exceptfds**, or **timeout** parameter points to a location outside of the process's allocated address space.

Related Information

In this book: "close, closex" in topic 1.2.48, "fcntl, flock, lockf" in topic 1.2.78, "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "msgctl" in topic 1.2.173, "msgget" in topic 1.2.174, "msgrcv" in topic 1.2.178, "msgsnd" in topic 1.2.180, "msgxrcv" in topic 1.2.181, "open, openx, creat" in topic 1.2.199, "read, readv, readx" in topic 1.2.224, "write, writex" in topic 1.2.330, "values.h" in topic 2.4.28, and Chapter 5, "Special Files."

1.2.243 semctl

Purpose
Controls semaphore operations.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (semid, semnum, cmd, val)
-- or --
int semctl (semid, semnum, cmd, buf)
-- or --
int semctl (semid, semnum, cmd, array)

int semid;
unsigned int semnum;
int cmd;
int val;
struct semid_ds *buf;
unsigned short array [ ];
```

Description
The **semctl** system call performs a variety of semaphore control operations as specified by the **cmd** parameter. The data type of the last parameter depends on the value of the **cmd** parameter. It is referred to as **val**, **buf**, or **array** to indicate one of the definitions given in the preceding **Syntax** section.

The first seven **cmd**s get and set the values of a **sem** structure, which is defined in the **sys/sem.h** header file and contains the following members:

```
ushort  semval;   /* Operation permission structure */
pid_t   sempid;   /* ID of last process that did a semop */
ushort  semncnt;  /* No. of processes awaiting semval > cval */
ushort  semzcnt;  /* No. of processes awaiting semval = 0 */
```

The following **cmd**s are executed with respect to the semaphore specified by the **semid** and **semnum** parameters.

GETVAL Returns the value of **semval**, if the current process has read permission.

SETVAL Sets the value of **semval** to the value specified by **val**, if the current process has write permission. When this **cmd** is successfully executed, the **semadj** value corresponding to the specified semaphore is cleared in all processes.

GETPID Returns the value of **sempid**, if the current process has read permission.

GETNCNT Returns the value of **semncnt**, if the current process has read permission.

GETZCNT Returns the value of **semzcnt**, if the current process has read

permission.

The following **cmd**s return and set every **semval** in the set of semaphores.

GETALL Stores **semval**s into the array pointed to by **array**, if the current process has read permission.

SETALL Sets **semval**s according to the array pointed to by **array**, if the current process has write permission. When this **cmd** is successfully executed, the semadj value corresponding to each specified semaphore is cleared in all processes.

The following **cmd**s are also available:

IPC_STAT Stores the current value of each member of the data structure associated with the **semid** parameter into the structure pointed to by **buf**, if the current process has read permission. This structure is defined in **sys/sem.h** and contains the following members:

```
struct ipc_perm sem_perm;  /* Operation permission structure */
struct sem  *sem_base;     /* Pointer to first semaphore in set */
ushort      sem_nsems;     /* Number of semaphores in the set */
time_t      sem_otime;     /* Time of last semop call */
time_t      sem_ctime;     /* Time of the last change to this */
                           /*  structure with a semctl call */
```

IPC_SET Sets the value of the following members of the data structure associated with the **semid** parameter to the corresponding value found in the structure pointed to by **buf**:

```
sem_perm.uid
sem_perm.gid
sem_perm.mode   /* Only the low-order nine bits */
```

This **cmd** can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of **sem_perm.uid** in the data structure associated with the **semid** parameter.

IPC_RMID Removes the semaphore identifier specified by the **semid** parameter from the system and destroys the set of semaphores and data structures associated with it. This **cmd** can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of **sem_perm.uid** in the data structure associated with the **semid** parameter.

Note: In a Transparent Computing Facility cluster, semaphores are not maintained across the cluster. This means that a process cannot communicate via semaphores to processes on another cluster site and the process itself cannot migrate.

Return Value

Upon successful completion, the value returned depends on the **cmd** parameter as follows:

cmd	**Return Value**
GETVAL	Returns the value of **semval**.

GETPID	Returns the value of **sempid**.
GETNCNT	Returns the value of **semncnt**.
GETZCNT	Returns the value of **semzcnt**.
All others	Return a value of 0.

If **semctl** fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **semctl** system call fails if one or more of the following are true:

EINVAL The **semid** parameter is not a valid semaphore identifier.

EINVAL The **semnum** parameter is less than 0 or greater than **sem_nsems**.

EINVAL The **cmd** parameter is not a valid command.

EACCES Operation permission is denied to the calling process.

ERANGE The **cmd** parameter is **SETVAL** or **SETALL** and the value to which **semval** is to be set is greater than the system-imposed maximum.

EPERM The **cmd** parameter is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal either to that of superuser or to the value of **sem_perm.uid** in the data structure associated with the **semid** parameter.

EFAULT The **buf** or **array** parameter points to a location outside of the process's allocated address space.

Related Information
In this book: "semget" in topic 1.2.244 and "semop" in topic 1.2.245.

1.2.244 semget

Purpose
Gets a set of semaphores.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key, nsems, semflg)
key_t key;
int nsems, semflg;
```

Description
The **semget** system call returns the semaphore identifier associated with the specified **key**. The **key** parameter is either the value **IPC_PRIVATE** or an IPC key constructed by the **ftok** subroutine (or by a similar algorithm). See " stdipc: ftok" in topic 1.2.284 for details about this subroutine. The **nsems** parameter specifies the number of semaphores in the set.

The **semflg** parameter is constructed by logically ORing one or more of the following values:

IPC_CREAT	Creates the data structure if it does not already exist.
IPC_EXCL	Causes the **semget** system call to fail if **IPC_CREAT** is also set and the data structure already exists.
S_IRUSR	Permits the process that owns the data structure to read it.
S_IWUSR	Permits the process that owns the data structure to modify it.
S_IRGRP	Permits the group associated with the data structure to read it.
S_IWGRP	Permits the group associated with the data structure to modify it.
S_IROTH	Permits others to read the data structure.
S_IWOTH	Permits others to modify the data structure.

The values that begin with **S_I**- are defined in the **sys/stat.h** header file and are a subset of the access permissions that apply to files.

The **semget** system call creates a data structure for the semaphore ID and an array containing **nsems** semaphores if one of the following is true:

- The **key** parameter is equal to **IPC_PRIVATE**.
- The **key** parameter does not already have a semaphore identifier associated with it, and **IPC_CREAT** is set.

Upon creation, the data structure associated with the new semaphore identifier is initialized as follows:

- **sem_perm.cuid** and **sem_perm.uid** are set equal to the effective user ID of the calling process.

sem_perm.cgid and **sem_perm.gid** are set equal to the effective group ID of the calling process.

The low-order nine bits of **sem_perm.mode** are set equal to the low-order nine bits of the **semflg** parameter.

sem_nsems is set equal to the value of the **nsems** parameter.

sem_otime is set equal to 0 and **sem_ctime** is set equal to the current time.

If the **key** parameter is not **IPC_PRIVATE**, **IPC_EXCL** is not set, and a semaphore identifier already exists for the specified **key**, then the value of the **nsems** parameter specifies the number of semaphores that the current process needs. If the **nsems** parameter is 0, then any number of semaphores is acceptable. If the **nsems** parameter is not 0, then the **semget** system call fails if the set contains fewer than **nsems** semaphores.

Note: In a Transparent Computing Facility cluster, semaphores are not maintained across the cluster. This means that a process cannot communicate via semaphores to processes on another cluster site and the process itself cannot migrate.

Return Value
Upon successful completion, a semaphore identifier is returned. If **semget** fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **semget** system call fails if one or more of the following are true:

EINVAL The **nsems** parameter is less than 0, equal to 0, or greater than the system-imposed limit.

EACCES A semaphore identifier exists for the **key** parameter but operation permission, as specified by the low-order nine bits of the **semflg** parameter, is not granted.

EINVAL A semaphore identifier exists for the **key** parameter, but the number of semaphores in the set associated with it is less than the value of the **nsems** parameter and the **nsems** parameter is not equal to 0.

ENOENT A semaphore identifier does not exist for the **key** parameter and **IPC_CREAT** is not set.

ENOSPC A semaphore identifier is to be created, but doing so would exceed the maximum number of identifiers allowed system wide.

EEXIST A semaphore identifier exists for the **key** parameter, and both **IPC_CREAT** and **IPC_EXCL** are set.

EAGAIN Cannot allocate space for a semaphore data structure.

Related Information
In this book: "semctl" in topic 1.2.243, "semop" in topic 1.2.245, and " stdipc: ftok" in topic 1.2.284.

1.2.245 semop

Purpose
Performs semaphore operations.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (semid, sops, nsops)
int semid;
struct sembuf sops [ ];
unsigned int nsops;
```

Description
The **semop** system call performs operations on the set of semaphores associated with the semaphore identifier specified by the **semid** parameter. The **sops** parameter points to an array of structures, each of which specifies a semaphore operation. The **nsops** parameter is the number of such structures in the array. The **sembuf** structure is defined in the **sys/sem.h** header file, and it contains the following members:

```
ushort  sem_num;  /* Semaphore number */
short   sem_op;   /* Semaphore operation */
short   sem_flg;  /* Operation flags */
```

Each semaphore operation specified by a **sem_op** is performed on the corresponding semaphore specified by **semid** and **sem_num**. The **sem_flg** for each operation is either 0 or constructed by logically ORing one or more of the following values:

SEM_UNDO Specifies whether to modify **semadj** values.

SEM_ORDER Specifies whether to perform the operations atomically or individually. (This applies only to the **sem_flg** of the first operation specified in the **sops** array.)

IPC_NOWAIT Specifies whether to wait or to return immediately when a semaphore's **semval** is not a certain value.

If **SEM_ORDER** is not set in **sops[0].sem_flg**, all of the semaphore operations specified in the **sops** array are performed atomically. This means that no **semval** value for any **sem_num** that appears in the entire array of operations is modified until all the semaphore operations can be completed. If the calling process must wait until some **semval** requirement is met, the **semop** system call does so before performing any of the operations. If any semaphore operation would cause an error to occur, none of the operations is performed.

If **SEM_ORDER** is set in **sops[0].sem_flg**, the operations are performed individually in the order that they appear in the **sops** array, regardless of whether any of the operations require the process to wait. If an operation encounters an error condition, the **semop** system call sets **SEM_ERR** in the **sem_flg** of the failing operation, sets **errno** to indicate the error, and returns a value of -1. In this case, the operations that

precede the failing one in the **sops** array have been performed, but those following it have not.

The action taken for **SEM_UNDO** and **IPC_NOWAIT** is described in the following text.

The **sem_op** field of the **sembuf** structure specifies one of the following three semaphore operations:

1. If **sem_op** is a positive integer and the current process has write permission, the value of **sem_op** is added to **semval**. If **SEM_UNDO** is set in **sem_flg**, the value of **sem_op** is also subtracted from the calling process's **semadj** value for the specified semaphore.

2. If **sem_op** is a negative integer and the current process has write permission, one of the following occurs:

   If **semval** is greater than or equal to the absolute value of **sem_op**, the absolute value of **sem_op** is subtracted from **semval**. Also, if **SEM_UNDO** is set in **sem_flg**, the absolute value of **sem_op** is added to the calling process's **semadj** value for the specified semaphore. The **exit** system call adds the **semadj** value to the semaphore's **semval** when the process terminates (see "exit, _exit" in topic 1.2.73).

   If **semval** is less than the absolute value of **sem_op** and **IPC_NOWAIT** is set in **sem_flg**, **semop** returns a value of -1 and sets **errno** to EAGAIN.

   If **semval** is less than the absolute value of **sem_op** and **IPC_NOWAIT** is not set in **sem_flg**, **semop** increments the **semncnt** associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:

   - **semval** becomes greater than or equal to the absolute value of **sem_op**. When this occurs, the value of **semncnt** associated with the specified semaphore is decremented, the absolute value of **sem_op** is subtracted from **semval** and, if **SEM_UNDO** is set in **sem_flg**, the absolute value of **sem_op** is added to the calling process's **semadj** value for the specified semaphore.

   - The **semid** for which the calling process is awaiting action is removed from the system (see "semctl" in topic 1.2.243). When this occurs, **errno** is set equal to EIDRM, and a value of -1 is returned.

   - The calling process receives a signal that is to be caught. When this occurs, the value of semncnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in the **signal** system call.

3. If **sem_op** is 0 and the current process has read permission, one of the following occurs:

   If **semval** is 0, **semop** returns a value of 0.

   If **semval** is not equal to 0 and **IPC_NOWAIT** is set in **sem_flg**, **semop** returns a value of -1 and sets **errno** to EAGAIN.

If **semval** is not equal to 0 and **IPC_NOWAIT** is not set in **sem_flg**, **semop** increments the **semzcnt** associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:

- **semval** becomes 0, at which time the value of **semzcnt** associated with the specified semaphore is decremented.
- The **semid** for which the calling process is awaiting action is removed from the system. When this occurs, **errno** is set equal to EIDRM, and a value of of -1 is returned.
- The calling process receives a signal that is to be caught. When this occurs, the value of **semzcnt** associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in the **signal** system call.

Note: In a Transparent Computing Facility cluster, semaphores are not maintained across the cluster. This means that a process cannot communicate via semaphores to processes on another cluster site and the process itself cannot migrate.

Return Value

Upon successful completion, the **semop** system call returns a value of 0. Also, the **sempid** value for each semaphore that is operated upon is set to the process ID of the calling process.

If **semop** fails, a value of -1 is returned, and **errno** is set to indicate the error. If **SEM_ORDER** was set in the **sem_flg** for the first semaphore operation in the **sops** array, **SEM_ERR** is set in the **sem_flg** for the failing operation.

Error Conditions

The **semop** system call fails if one or more of the following are true for any of the semaphore operations specified by the **sops** parameter. If the operations were performed individually, see the preceding discussion of **SEM_ORDER** for more information about error situations.

EINVAL The **semid** parameter is not a valid semaphore identifier.

EFBIG **sem_num** is less than 0 or it is greater than or equal to the number of semaphores in the set associated with the **semid** parameter.

E2BIG The **nsops** parameter is greater than the system-imposed maximum.

EACCES Operation permission is denied to the calling process.

EAGAIN The operation would result in suspension of the calling process, but **IPC_NOWAIT** is set in **sem_flg**.

ENOSPC The limit on the number of individual processes requesting a **SEM_UNDO** would be exceeded.

EINVAL The number of individual semaphores for which the calling process requests a **SEM_UNDO** would exceed the limit.

ERANGE An operation would cause a **semval** to overflow the system-imposed limit.

EAGAIN Space cannot be allocated for an internal data structure (**SEM_UNDO**).

EAGAIN Insufficient kernel memory available to allocate the semaphore data structures.

ERANGE An operation would cause a **semadj** value to overflow the system-imposed limit.

EFAULT The **sops** parameter points to a location outside of the process's allocated address space.

EINTR The **semop** system call received a signal.

EIDRM The semaphore identifier **semid** has been removed from the system.

Related Information
In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, "exit, _exit" in topic 1.2.73, " fork, vfork" in topic 1.2.83, "semctl" in topic 1.2.243, and "semget" in topic 1.2.244.

1.2.246 send, sendto, sendmsg

Purpose
Sends a message from a socket.

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>

int send (s, msg, len, flagsint sendto (s, msg, len, flags, to
int s;                      int s;
char *msg;                  char *msg;
int len, flags;             int len, flags;
                            struct sockaddr *to;
int sendmsg (s, msg, flags) int tolen;
int s;
struct msghdr msg [ ];
int flags;
```

Description

The **send** system call sends a message only when the socket is in a connected state. The **sendto** and **sendmsg** system calls can be used at any time.

Give the address of the target with **to**, specifying its size with **tolen**. Specify the length of the message with **len**. Specify the socket ID as returned by the **socket** system call with **s**. If the message is too long to pass through the underlying protocol, the error **EMSGSIZE** is returned and the message is not transmitted.

No indication of failure to deliver is implied in a **send**. Return values of -1 indicate some locally detected errors.

If no space for messages is available at the sending socket to hold the message to be transmitted, the **send** system call blocks unless the socket is in a nonblocking I/O mode.

Use the **select** system call to determine when it is possible to send more data.

The **flags** argument to send a call is formed by logically ORing one or both of the values shown in the following list:

MSG_OOB Processes out-of-band data on sockets that support this notion, for instance, **SOCK_STREAM**. The underlying protocol must also support out-of-band data.

MSG_DONTROUTE Sends without using routing tables; this is usually used only by diagnostic or routing programs.

For a description of the **msghdr** structure, see "recv, recvfrom, recvmsg" in topic 1.2.227. If the Transparent Computing Facility is installed, processes using socket-related system calls are limited to operating on a single cluster site.

Return Value

Upon successful completion, the number of characters sent is returned. If the **send**, **sendto**, or **sendmsg** system call fails, a value of -1 is returned, and **errno** is set to indicate the error. A socket marked with the **O_NDELAY** flag returns ZERO in the situation where it would otherwise have blocked.

Error Conditions
The system call fails if one or more of the following are true:

EBADF The **s** parameter is not valid.

ENOTSOCK The **s** parameter refers to a file, not a socket.

EFAULT The **addr** parameter is not in a readable part of the user address space.

EMSGSIZE The socket requires that the message be sent all at once, and the message is too large for that to happen.

EAGAIN The socket is marked nonblocking with the **O_NONBLOCK** flag or the **FIONBIO ioctl** in the situation where it would otherwise have blocked.

ENOBUFS The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.

ENOBUFS The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

Related Information
In this book: "recv, recvfrom, recvmsg" in topic 1.2.227 and "socket" in topic 1.2.275.

1.2.247 setbuf, setvbuf

Purpose
Assigns buffering to a stream.

Library
Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;

int setvbuf (stream, buf, mode, size)
FILE *stream;
char *buf;
int mode;
size_t size;
```

Description
The **setbuf** and **setvbuf** functions override the default buffering on the specified stream. These functions may be used only after the stream pointed to by *stream* has been associated with an open file and before any other operation is performed on the stream.

If **setbuf** and **setvbuf** are not called prior to the first time the *stream* is read or written, a buffer will be allocated automatically. For *streams* directed to terminals, this will enable line buffering; except for **stderr** which is unbuffered by default.

The **setbuf** subroutine causes the user supplied buffer *buf* to be used instead of the default buffer. This buffer must be of size BUFSIZ characters, as defined in <**stdio.h**>. If the *buf* parameter is a NULL pointer, the *stream* will be unbuffered.

For the **setvbuf** function, **mode** determines how *stream* is buffered:

_IOFBF Causes input/output to be fully buffered.

_IOLBF Causes input/output to be line buffered. The buffer is flushed when a new line is written, the buffer is full, or input is requested.

_IONBF Causes input/output to be completely unbuffered.

For fully buffered and line buffered streams, a non-null *buf* parameter may be specified. It must point to an array of *size* characters to be used for the buffering; or a null *buf* parameter may be specified to request that **setvbuf**, allocate its own buffer of size *size*. A good value for *size* is the constant BUFSIZ defined in <**stdio.h**>. If input/output is unbuffered, the *buf* and *size* parameters are ignored.

Note: A common source of error is allocating buffer space as an automatic variable in a code block and then failing to close the stream in the same block.

Error Conditions

The **setvbuf** function fails if the following is true:

EBADF The file descriptor underlying **stream** is not valid.

Related Information

In this book: "fopen, freopen, fdopen" in topic 1.2.82, "getc, fgetc, getchar, getw, getwc, fgetwc, getwchar" in topic 1.2.91, "malloc, free, realloc, calloc, valloc, alloca, mallopt, mallinfo" in topic 1.2.162, "putc, putchar, fputc, putw, putwc, putwchar, fputwc" in topic 1.2.213, "setbuffer, setlinebuf" in topic 1.2.248, and " stdio" in topic 1.2.283.

1.2.248 setbuffer, setlinebuf

Purpose
Assigns buffering to a stream.

Library
Standard I/O Library (**libc.a**)

Syntax

```
void setbuffer (stream, buf,voidesetlinebuf (stream)
FILE *stream;               FILE *stream;
char *buf;
size_t size;
```

Description
The **setbuffer** subroutine, an alternate form of the **setbuf** subroutine described on page 1.2.247, is used after **stream** has been opened, but before it is read or written. The character array **buf**, whose size is determined by the **size** parameter, is used instead of an automatically allocated buffer. If the **buf** parameter is a null character pointer, input/output is completely unbuffered.

The **setbuffer** subroutine is not needed under normal circumstances since the default file I/O buffer size are optimal.

The **setlinebuf** subroutine is used to change **stdout** or **stderr** from block buffered or unbuffered to line buffered. Unlike the **setbuf** and **setbuffer** subroutines, the **setlinebuf** subroutine can be used any time the file descriptor is active.

For more information on stream buffering, see "setbuf, setvbuf" in topic 1.2.247.

Related Information
In this book: "fclose, fflush" in topic 1.2.77, "fopen, freopen, fdopen" in topic 1.2.82, "fread, fwrite" in topic 1.2.84, "getc, fgetc, getchar, getw, getwc, fgetwc, getwchar" in topic 1.2.91, "malloc, free, realloc, calloc, valloc, alloca, mallopt, mallinfo" in topic 1.2.162, "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf" in topic 1.2.208, "putc, putchar, fputc, putw, putwc, putwchar, fputwc" in topic 1.2.213, "puts, fputs, putws, fputws" in topic 1.2.216, and "setbuf, setvbuf" in topic 1.2.247.

1.2.249 setgroups

Purpose
Sets the group access list.

Syntax

```
#include <grp.h>

int setgroups (ngroups, gidset);
int ngroups, *gidset;
```

Description
The **setgroups** system call sets the group access list of the current user process according to the array pointed to by the **gidset** parameter. The **ngroups** parameter indicates the number of entries in the array and must not be more than **NGROUPS**, as defined in the **grp.h** header file. Only a process with an effective user ID of superuser can set new groups.

Return Value
Upon successful completion, a value of 0 is returned. If the **setgroups** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **setgroups** system call fails if one or more of the following is true:

EPERM The effective user ID of the calling process is not superuser.

EINVAL The value of the **ngroups** parameter is greater than **NGROUPS**.

EFAULT The **gidset** parameter points to a location outside of the process's allocated address space.

Related Information
In this book: "getgroups" in topic 1.2.97 and "initgroups" in topic 1.2.135.

1.2.250 setjmp, longjmp, _setjmp, _longjmp

Purpose
Saves and restores the current execution context.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <setjmp.h>


int setjmp (ctxt)                    void longjmp (ctxt, val)
jmp_buf ctxt;                        jmp_buf ctxt;
                                     int val;
int _setjmp (ctxt)
jmp_buf ctxt;

void _longjmp (ctxt, val)
jmp_buf ctxt;
int val;
```

Description
The **setjmp** and **longjmp** subroutines can be useful when handling errors and interrupts encountered in low-level subroutines of a program.

The **setjmp** subroutine saves the current stack context and signal mask in the buffer specified by the **ctxt** parameter. The **setjmp** subroutine returns a value of 0.

The **longjmp** subroutine restores the stack context and signal mask that were saved by the **setjmp** subroutine in the corresponding **ctxt** buffer. After the **longjmp** subroutine has completed, the program execution continues as if the corresponding call to **setjmp** had just returned the value of the **val** parameter. The subroutine that called **setjmp** must not have returned before the completion of the **longjmp** subroutine. **Setjmp** and **longjmp** save and restore the signal mask **sigmask**, while **_setjmp** and **_longjmp** manipulate only the stack context.

The **longjmp** subroutine cannot return 0 to the previous context. The value 0 is reserved to indicate the actual return from the **setjmp** subroutine when first called by the program. If the **longjmp** subroutine is passed a **val** parameter of 0, then execution continues as if the corresponding call to the **setjmp** subroutine had returned a value of 1. All global and static data have values as of the time the **longjmp** subroutine is called.

Warning: If the **longjmp** subroutine is called with a **ctxt** parameter that was not previously set by **setjmp**, or if the subroutine that made the corresponding call to **setjmp** has already returned, then the results of the **longjmp** subroutine are undefined.

Related Information
In this book: "sigaction, sigvec, signal" in topic 1.2.263.

1.2.251 setlocale

Purpose
Sets program's locale.

Syntax

#include <locale.h>

char *setlocale (int category, const char *locale);

Description

The **setlocale** subroutine selects the appropriate portion of the program's locale as specified by the category and locale arguments. The **setlocale** subroutine may be used to change or query the program's entire current locale or portions thereof. The value **LC_ALL** for category names the program's entire locale. **LC_COLLATE** affects the behavior of the **strcoll** and **strxfrm** subroutines. **LC_CTYPE** affects the behavior of the character handling subroutines and the multibyte functions. **LC_MONETARY** affects the monetary formatting information returned by the **localeconv** subroutine. **LC_NUMERIC** affects the decimal-point character for the formatted input/output subroutines and the string conversion subroutines, as well as the non-monetary formatting information returned by the **localeconv** subroutine. **LC_TIME** affects the behavior of the **strftime** subroutine.

A value of **C** for locale specifies the minimal environment for **C** translation; a value of **" "** for locale specifies the implementation defined native environment. Other implementation defined strings may be passed as the second argument to **setlocale**.

At program startup, the equivalent of **setlocale(LC_ALL,"C");** is executed.

The implementation will behave as if no library subroutine calls the **setlocale** subroutine.

Options and Affected Areas

Subtopics
1.2.251.1 The category Option
1.2.251.2 The locale Option

1.2.251.1 The category Option

Below is the list of parameters which **setlocale** understands.

Category	**Action**
LC_ALL	All the categories.
LC_COLLATE	Loads in new collation table that determines the ordering of characters.
LC_CTYPE	Loads in new collation table to be used for character examination functions.
LC_MONETARY	Update the structure that hold various monetary standard information, position of sign and the characters for the monetary standards.
LC_NUMERIC	The type of decimal point used.
LC_TIME	The format of time and weekday and the corresponding words for them.
LC_MESSAGE	The language in which system error messages are to be shown in.

Functions affected as a result of calling setlocale.

Category	Functions affected.		
LC_ALL	All functions mentioned below.		
LC_COLLATE	_NCxcol()	NCcollate()	strxfrm()
	_NLxcol()	NCcoluniq()	wcscmp()
	mbscoll()	NCcolval()	wcsncmp()
	mbscmp()	NCeqvmap()	wcscoll()
	mbsncmp()	strcoll()	
LC_CTYPE	_tolower()	ispunct()	iswlower()
	_toupper()	isshift()	iswprint()
	isalnum()	isspace()	iswpunct()
	isalpha()	isupper()	iswspace()
	iscntrl()	iswalnum()	iswupper()
	isgraph()	iswalpha()	iswxdigit()
	islower	iswcntrl()	mblen()
	isprint()	iswdigit()	mbsadvance()
	mbschr()	iswgraph()	mbscat()
	mbscmp()	NCisgraph()	NLstrncat()
	mbscpy()	NCislower()	NLstrncmp()
	mbscspn()	NCisNLchar()	NLstrncpy()
	mbsinvalid()	NCisprint()	NLstrpbrk()
	mbslen()	NCispunct()	NLstrrchr()
	mbsncat()	NCisshift()	NLstrspn()
	mbsncmp()	NCisspace()	NLstrtok()

	mbsncpy()	NCisupper()	NCisxdigit()
	mbspbrk()	NCstrcat()	tolower()
	mbsrchr()	NCstrchr()	toupper()
	mbsspn()	NCstrcpy()	towlower()
	mbstok()	NCstrcspn()	towupper()
	mbstomb()	NCstrlen()	wcscat()
	mbstowcs()	NCstrncat()	wcschr()
	mbtowc()	NCstrncmp()	wcscmp()
	NCchrlen()	NCstrncpy()	wcscpy()
	NCdec()	NCstrpbrk()	wcscspn()
	NCdechr()	NCstrrchr()	wcslen()
	NCdecode()	NCstrspn()	wcsncat()
	NCdecstr()	NCstrtok()	wcsncmp()
	NCenc()	NLchrlen()	wcsncpy()
	NCencode()	NLstrcat()	wcspbrk()
	NCencstr()	NLstrchr()	wcsrchr()
	NCisalnum()	NLstrcpy()	wcsspn()
	NCisalpha()	NLstrcspn()	wcstok()
	NCiscntrl()	NLstrdlen()	wctombs()
	NCisdigit()	NLstrlen()	wctomb()
LC_MONETARY	localeconv.		
LC_NUMERIC	atof, fprintf, localeconv, printf, scanf, sprintf, ssc		
LC_TIME	NLstrtime, NLtmtime, strftime.		
LC_MESSAGE	All message accessing routines.		

Note: Many of the above function/macros will affect other library routines such as **regex**, but since their dependence is solely through the above interface, they will not be mentioned separately.

1.2.251.2 The locale Option

char *setlocale (int category, char *locale);

The variable **locale** consists of two parts. First, the locale which is the name of the configuration file and second, the modifier. The modifier is separated from the filename, using the **@**, or "at" sign. The filename, which will follow the standards set in the Version 3 document, is **language_territory.codeset**. In reality, this is the name of a configuration file that contains keywords corresponding to various elements of **setlocale**. If the filename included in the locale variable points to an absolute path, the given absolute path will be read. On the other hand, if the filename does not start with a "/", **setlocale** will look in the environment for a variable called **MBCS_CFG_DIR**. This variable will contain the path to directory in which the directory named **locale.dir** resides. **setlocale** processes the file pointed to by **$MBCS_CFG_DIR/locale.dir/<filename>** and interprets its various options. In the event the **MBCS_CFG_DIR** is not set, the hard-coded default file **/usr/lib/mbcs.cfg.dir/locale.dir/<filename>** is opened for reading.

The contents of the configuration file is a number of tokens indicating the various options of the subroutines affected by the call to **setlocale**. The format is as follows:

```
#####
# Comments et.al.
#
MBDATE=MM/DD/YY
MBSDAY=Sun:Mon:Tue:Wed:Thu:Fri:Sat
```

Note: Some fields such as **MBSDAY** contain an array of values separated by a **:**, or colon; if you need a colon as a literal and not a field separator, then it must be escaped using a backslash character. Multiple definitions can be made on a line, provided the definitions are separated by a semi-colon. There are no spaces between the equal equal sign, and the variable name and its value; the double quote character, has no special meaning and it will be interpreted as a part of the variable name/value. If a token is defined and is not legal or does not belong to the current category, it is ignored.

Below is a list of understood tokens, their groups and their subroutines:

Tokens and their respective Category

Name	Category	Functions Effected
MBDATE	LC_TIME	
MBLANG	LC_MESSAGE	NLgetmsg, catopen
MBLDATE	LC_TIME	strftime(%x)
MBLDATIM	LC_TIME	strftime(%c)

MBLDAY	LC_TIME	strftime(%A)
MBSDAY	LC_TIME	strftime(%a)
MBLMONTH	LC_TIME	strftime(%B)
MBSMONTH	LC_TIME	strftime(%b)
MBTIME	LC_TIME	strftime(%X)
MBAM_STR	LC_TIME	strftime(%p)
MBPM_STR	LC_TIME	strftime(%p)
MBCURSTR	LC_MONETARY	nl_langinfo
MBNUMSEP	LC_NUMERIC	printf, scanf, atof
MBTMISC	LC_TIME	Unix "at" command
MBMONEY	LC_MONETARY	localeconv
MBTSTRS	LC_TIME	
MBNOSTR	LC_MESSAGE	
MBYESSTR	LC_MESSAGE	
MBCOLTAB	LC_COLLATE	See above for a list
MBCTYPE	LC_CTYPE	See above for a list
MBCONV	LC_ALL	All wide char stdio functions

The modifier can be used to modify the behavior of various subroutines of **setlocale**. For instance, if you wish to use the **GR_SW.pc850** (German spoken in Switzerland) locale, but you wish to use the *am, pm* strings belonging to the time subroutine with different values than the ones

defined in the **GR_SW.pc850** locale, you may do this as follows:

```
setlocale (LC_ALL, "Gr_SW.pc850@MBAM_STR=am;MBPM_STR=pm");
```

Note that no space is used before and after the equal signs, the semicolons and the '@' modifier when calling **setlocale**.

There are two other values recognized for the locale value. If locale is a NULL pointer, the user will be given a **char** pointer containing all the information about the current locale for the category inquired. This pointer can be used at a later call to **setlocale** to restore the locale that was surveyed last, provided that the contents of memory pointed to by the **char** pointer, which is passed to the user, was saved (the contents of the array could change by calling **setlocale**, before trying to restore the locale).

Example

```
old_locale = setlocale (LC_TIME, (char *)NULL);
   setlocale(LC_TIME, "Fr_FR.pc850");
   setlocale(LC_TIME, old_locale);
```

The above example will fail. The correct format will look like this:

```
old_locale = setlocale(LC_TIME, (char *)NULL);
   cspace = malloc(strlen(old_locale+1));
   strcpy(cspace, old_locale);
   setlocale(LC_TIME, "Fr_FR.pc850");
   setlocale(LC_TIME, cspace);
   free(cspace);
```

The other value recognized for **setlocale** is a pointer to a NULL. This means to set the locale to the default locale. This is established by examining the environment value for the category at hand; in other words, the program will act as if you passed the environment variable for that category, as follows:

```
setlocale(LC_TIME, ""); /* is equivalent to: */
setlocale(LC_TIME, getenv("LC_TIME"));
```

If one calls **setlocale** in the form

```
setlocale(LC_ALL, "C");
```

all the options will be reset to their **C** locale value. This is the default value at start-up. All programs at start-up will act if they called **setlocale(LC_ALL, "C");** this is achieved by having all the structures used by **setlocale** initialized to the **C** locale.

The three tokens **MBCONV, MBCTYPE,** and **MBCOLTAB** are different, in the sense that they involve loading an external collation or conversion table. **setlocale** has the two hard-coded tables, one for a **C** locale conversion table, and a second for a **C** locale collation/character type table. There are a few advantages in having the two tables hard-coded. First, there will be no need for loading the two collation and conversion tables at the beginning of the execution. This increases the performance and reliability of the system, since a damage to the **C** locale collation or conversion table will break all the executables relying on tables. Second, the hard-coded tables will be a part of the text of the executables, and in an AIX environment, executables running simultaneously

share their text.

The values of **MBCONV, MBCTYPE,** and **MBCOTAB** will be understood as filenames to be loaded and placed in the appropriate structures, except in the case where the value is equal to **C**; in this case, the assumption will be made that the user wants the **C** locale, and the appropriate table pointers to their hard-coded **C** locale values will be set.

Return Value

If a pointer to a string is given for locale and the selection can be honored, the **setlocale** subroutine returns a pointer to the string associated with the specified category for the new locale. If the selection cannot be honored, the **setlocale** subroutine returns a null pointer and the program's locale is not changed.

Parameters passed to **setlocale** update the working environment (this is not the same as the environment variables) of that particular process, such as its monetary representation, and some of this information is passed to the kernel. **setlocale** is responsible for loading in the conversion table using **mbgettabl** and the collation table.

Related Information
In this book: "ctype" in topic 1.2.55, "langinfo.h" in topic 2.4.10, "locale.h" in topic 2.4.12, "nl_langinfo" in topic 1.2.198, "string" in topic 1.2.288, "strcoll, strncoll, strxfrm, mbscoll, mbsncoll, wcscoll, wcsncoll" in topic 1.2.286, "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf" in topic 1.2.208, and "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf, wsscanf" in topic 1.2.241.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.252 setpgid, setpgrp, setsid

Purpose
Sets the process group or session ID of a process.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>

int setpgid (pid pgid)
pid_t pid, pgid;

int setsid ()

int setsid ()
```

Description
The **setpgid** system call is used to either join an existing process group or to create a new process group. Upon successful completion, the process group ID of the process specified by the **pid** parameter is set to the pgid parameter. If the *pid* parameter is 0, the process group ID of the calling process is set to *pgid*. If the *pgid* parameter is 0, the process group ID is set equal to the process ID of the process specified by *pid*. Only child processes of the calling process in the same process session are valid targets for the **setpgid** call. For a session leader, the process ID value, the process group ID value, and the session ID value of the process must always be equal. Therefore, **setpgid** does not allow the process group ID of a session leader to change.

The **setsid** system call is used to create a new session. It may not be called by a process which is either a session leader or a process group leader. The calling process becomes the session leader of this new session and upon return has no controlling terminal. The process group ID of the calling process is set equal to its process ID. The calling process becomes the only process in the new process group and the only process in the new session.

The **setpgrp** system call is provided for compatibility with UNIX System V systems. This system call is equivalent to **setsid**, creating a new session and establishing the calling process as a session leader and process group leader.

Compatibility Note
For compatibility with 4.3BSD systems, there is a second **setpgrp** routine available in the Berkeley Compatibility Library (**libbsd.a**):

```
int setpgid (pid, pgid)
pid_t pid, pgid;
```

This routine is equivalent to **setpgid**(), allowing a process to either join an existing process group or to create a new process group.

Return Value
Upon successful completion, **setpgid** returns a value of 0 and **setsid** returns the process group ID value of the calling process. Otherwise, -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **setpgid** system call fails if any of the following is true:

ESRCH The process specified by the **pid** parameter does not exist.

EINVAL The value of the **pgid** argument is less than 0.

EPERM The process indicated by the **pid** argument is a session leader.

The value of the **pid** argument is valid but matches the process ID of a child process of the calling process, and the child process is not in the same session as the calling process.

The value of the **pgid** argument does not match the process ID of the process indicated by the **pid** argument, and there is no process with a process group ID that matches the value of the **pgid** argument in the same session as the calling process.

ESRCH The value of the **pid** argument does not match the process ID of the calling process or of a child process of the calling process.

The **setsid** system call fails if the following is true:

EPERM The calling process is already a session or process group leader.

Related Information
In this book: " fork, vfork" in topic 1.2.83, "getpid, getpgrp, getppid" in topic 1.2.110.

1.2.253 setquota

Purpose
Enables or disables quotas on a file system.

Syntax

int setquota (**special, file**)
char *__**special, *file**__;

Description
Disk quotas are enabled or disabled with the **setquota** system call. The **special** parameter indicates a block special device on which a mounted file system exists. If the **file** parameter is not NULL, it specifies a file in that file system from which to take the quotas. If the **file** parameter is NULL, quotas are disabled on that file system. The quota file must exist; it is normally created with the **quotacheck** program. Only the superuser may turn quotas on or off.

Return Value
A 0 return value indicates that the call succeeded. A return value of -1 indicates that an error occurred, and an error code is stored in the global variable **errno**.

Error Conditions
The possible errors are:

EINVAL The system is not configured to support the **QUOTA** option.

EPERM The command requires privilege, and the calling process's effective user ID was not superuser.

ENODEV The device associated with **special** does not exist.

ENOTBLK The device associated with **special** is not a block device.

ENXIO The device associated with **special** does not exist.

EROFS The file specified by **file** resides on a read-only file system.

EACCES The file specified by **file** resides on a file system different from **special**.

EACCES The file specified by **file** is not a plain file.

EIO An I/O error occurred while accessing the file specified by **file**.

The following errors are applicable to any system call which requires path name resolution:

ENOTDIR A component of the path prefix is a not a directory.

ENOENT A component of the path prefix does not exist.

EACCES Search permission is denied on a component of the path prefix.

ENOENT The path name is null.

ENAMETOOLONG

A component of a path name exceeds 255 characters, or an entire path name exceeds 1023 characters.

ESTALE The process's root or current directory is located in an NFS virtual file system that has been unmounted.

EFAULT The **special** or **file** parameter points to a location outside of the process's allocated address space.

ELOOP Too many symbolic links were encountered in translating the path name.

EIO An I/O error occurred during the operation.

If the Transparent Computing Facility is installed on your system, **setquota** can also fail if one or more of the following are true:

ESITEDN1 Either *special* or *file* cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE *special* or *file* is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of *special* or *file* is replicated but is not stored on any site which is currently up.

EROFS *file* resides on a replicated file system in which the primary copy is unavailable.

ENLDEV *special* is a non-TTY character special file which corresponds to a device physically attached to another site in the cluster.

EINTR A signal was caught during the system call.

Related Information

In this book: "getrlimit, setrlimit, vlimit" in topic 1.2.115, "getrusage, vtimes" in topic 1.2.116, "ulimit" in topic 1.2.313, and "quota" in topic 1.2.218.

The description of **quotacheck** and **quotaon** in *AIX Operating System Commands Reference*.

1.2.254 setreuid, setregid

Purpose
Sets real and effective user and group IDs.

Library
Standard C Library (**libc.a**)

Syntax

```
int setreuid (ruid, euid)
int ruid, euid;

int setregid (rgid, egid)
int rgid, egid;
```

Description
The **setreuid** subroutine sets the real and effective user ID of the current process according to the parameters **ruid** and **euid**. If either of these parameters equals -1, the current user ID is used.

Any process can change the real user ID or the effective user ID to the other, but only a process with an effective user ID of superuser can make other kinds of changes.

The **setregid** subroutine sets the real and effective group ID of the current process according to the parameters **rgid** and **egid**. If either of these parameters equals -1, the current group ID is used.

Any process can change the real group ID or the effective group ID to the other, but only a process with an effective user ID of superuser can make other kinds of changes.

If you need the **setruid** or **seteuid** functions, they can be constructed as:

```
#define seteuid(id) setreuid(-1, (id))
#define setruid(id) setreuid((id), -1)
```

Return Value
When the call succeeds, a value of 0 is returned. If the **setreuid** or **setregid** subroutines fail, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions

EPERM The current process does not have an effective user ID of superuser. This process attempted to change other than from the effective user (group) ID to the real user (group) ID, or vice versa.

Related Information
In this book: "getuid, geteuid, getgid, getegid" in topic 1.2.124 and "setuid, setgid" in topic 1.2.255.

1.2.255 setuid, setgid

Purpose
Sets a process's user and group IDs.

Syntax

#include <sys/types.h>

int setuid (uid)	**int setgid (gid)**
uid_t uid;	**gid_t gid;**

Description
The **setuid** system call sets the real user ID, the effective user ID, and the saved-set-user ID of the calling process. If the effective user ID of the calling process is superuser, then the real user ID, effective user ID, and the saved-set-user ID are set to the value of the **uid** parameter. If the effective user ID of the calling process is not superuser, but the real user ID or the saved-set-user ID is equal to the value of the **uid** parameter, then the effective user ID is set to the value of the **uid** parameter.

The **setgid** system call sets the real group ID, the effective group ID, and the saved-set-group ID of the calling process. If the effective user ID of the calling process is superuser, then the real group ID, effective group ID, and the saved-set-group ID are set to the value of the **gid** parameter. If the effective user ID of the calling process is not superuser, but the real group ID or the saved-set-group ID is equal to the value of the **gid** parameter, then the effective group ID is set to the value of the **gid** parameter.

Return Value
Upon successful completion, a value of 0 is returned. If the **setuid** or **setgid** system call fails, then a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **setuid** and **setgid** system calls fail if the following is true:

EPERM The **uid** (**gid**) parameter is not equal to the real user (group) ID of the process or to the saved-set-user (group) ID as set by the **exec** system call, and the effective user ID is not superuser.

EINVAL The **uid** (**gid**) parameter is not a valid user (group) ID.

Related Information
In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, "getpid, getpgrp, getppid" in topic 1.2.110, "getuid, geteuid, getgid, getegid" in topic 1.2.124, " run: runl, runv, runle, runve, runlp, runvp" in topic 1.2.239, and "rexec" in topic 1.2.235.

1.2.256 setxuid

Purpose
Uses real user ID or group ID on subsequent invocations of the **exec** or **run** system calls.

Syntax

```
#include <sys/types.h>

int setxuid(options)
int options;

#define GS_XUID 1
#define GS_XGID 2
```

Description
The **setxuid** system call allows a process to restore the real user ID and/or real group ID to be used as the effective user ID and/or effective group ID, respectively, at the end of a later **exec**, **rexec** or **run** system call. **setxuid** can turn these options on or off. The possible options are:

GS_XUID If this bit is turned on, the effective user ID of invoked programs is the current process's real user ID, unless the invoked program already has its **setuid** mode bit set. If this bit is turned off, the effective user ID follows normal semantics; it is not modified in an invoked program unless the program has its **setuid** mode bit set.

GS_XGID If this bit is turned on, the effective group ID of invoked programs is the current process's real group ID, unless the invoked program already has its **setgid** mode bit set. If this bit is turned off, the effective group ID follows normal semantics; it is not modified in an invoked program unless the program has its **setgid** mode bit set.

This system call permits a **setuid** program to invoke another program with its caller's permission while using the **setuid** program's permission to determine execute access. Its use in the invoking process is equivalent to using the **setuid** and/or **setgid** system call as the first instructions in the invoked process unless the invoked program has its **setuid** and/or **setgid** mode bits set.

The effect of this operation is carried around with the process until reset by another call to **setxuid** or until a new image has been loaded by a successful **exec**, **rexec** or **run** system calls. In particular, this state is inherited by all child processes.

A typical use of **setxuid** is to limit when or by whom another program can be run. In place of a program whose use is to be restricted, a small program of the same name can be installed. This program uses **setuid** or **setgid** for a more privileged user or group. After determining that the user should be allowed to run the actual program, it calls **setxuid** and then calls **exec** to execute the program, which has been installed elsewhere with restricted permissions.

Return Value
The old value of the options is returned.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, " rexec: rexecl, rexecv, rexecle, rexecve, rexeclp, rexecvp" in topic 1.2.236, " run: runl, runv, runle, runve, runlp, runvp" in topic 1.2.239, and "setuid, setgid" in topic 1.2.255.

1.2.257 sfent, sfnum, sfname, sfctype, sfxcode, setsf, endsf

Purpose
Site file entry access routines.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sf.h>
#include <sys/types.h>


struct sf *sfent()              struct sf *sfxcode(xcode)
                                short xcode;
struct sf *sfnum(sitenum)
siteno_t sitenum;               setsf()

struct sf *sfname(sitename) endsf()
char *sitename;
struct sf *sfctype(cputype)
char *cputype;
```

Description
These routines are used to access the site file **/etc/site**. The site file is used primarily by the Transparent Computing Facility and contains information about each machine in the TCF cluster. If TCF is not installed, the site file has one entry for the local machine. This entry contains the name of the machine as entered during the installation of the AIX Operating System. This name is also returned in **nodename** by the **uname** system call.

The **sfent**, **sfnum**, **sfname**, **sfctype** and **sfxcode** subroutines return a pointer to a structure containing the fields of a line in the **/etc/site** file. The **sf** structure is defined in the **sf.h** header file and contains the following members:

```
short   sf_id;          /* Site ID                */
char    *sf_sname;      /* Site name              */
char    *sf_local;      /* Local file system name */
char    *sf_ctype;      /* CPU type name          */
short   sf_ccode;       /* CPU type code          */
char    *sf_cname;      /* Full CPU type name     */
char    *sf_fname;      /* Full site name         */
short   sf_speed;       /* CPU speed              */
```

The **setsf** subroutine opens the site file and keeps it open across the other calls. If several entries are to be accessed, **setsf** should be called first. If the site file is already open, **setsf** rewinds it.

The **sfent** subroutine reads the next entry from the site file **/etc/site**. The site file is opened if necessary.

The **sfnum** subroutine returns the site information for the machine specified by **sitenum**. If the specified site number is not found, NULL is returned. If the site file had to be opened to satisfy this request, it is also closed.

The **sfname** subroutine returns the site information for the machine specified by **sitename**. If the specified site name is not found, NULL is returned. Case is ignored in name comparisons. If the site file had to be opened to satisfy this request, it is also closed.

The **sfctype** subroutine returns the site information for the next entry with a CPU type name that matches the type specified by **cputype**. NULL is returned if a matching entry is not found. The site file is opened if necessary.

The **sfxcode** subroutine returns the site information for the next entry with a CPU type code that matches the type specified by **xcode**. NULL is returned if a matching entry is not found. The site file is opened if necessary.

The **endsf** subroutine closes the site file.

Return Value
All information is returned in a static area, so it must be copied if it is to be saved. All routines return NULL upon error.

File

/etc/site Site description file

Related Information
In this book: "uname, unamex" in topic 1.2.316.

1.2.258 shmat

Purpose
Attaches a shared memory segment to the current process.

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr;
int shmflg;
```

Description
The **shmat** system call attaches the shared memory segment associated with the shared memory identifier (returned by **shmget**) specified by the **shmid** parameter to the address space of the calling process.

The segment is attached at the address specified by the **shmaddr** parameter as follows:

If the **shmaddr** parameter is equal to 0, the segment is attached at the first available address as selected by the system.

If the **shmaddr** parameter is **not** equal to 0, and **SHM_RND** is set in **shmflg**, the segment is attached at the next lower segment boundary. This address given by (**shmaddr** - (**shmaddr** module **SHMLBA**)).

If the **shmaddr** parameter is **not** equal to 0 and **SHM_RND not** set in **shmflg**, the segment is attached at the address given by the **shmaddr** parameter. If this address does not point to a segment boundary, then the **shmat** system call returns the value -1 and sets **errno** to EINVAL.

The **shmflg** parameter specifies several options. Its value is either 0, or is constructed by logically ORing one or more of the following values:

SHM_RND Rounds the address given by the **shmaddr** parameter to the next lower segment boundary, if necessary.

SHM_RDONLY Specifies read-only mode instead of the default read-write mode.

The shared memory segment is attached for reading if **SHM_RDONLY** is set in **shmflg** and if the current process has read permission. If **SHM_RDONLY** is not set and the current process has both read and write permission, then it is attached for reading and writing.

Note: In a Transparent Computing Facility cluster, shared memory segments exist only on the cluster site on which they are created. Consequently, processes that use shared memory segments cannot be migrated to other sites and processes on different cluster sites cannot share memory segments.

Return Value
Upon successful completion, the segment start address of the attached shared memory segment is returned. If **shmat** fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **shmat** system call fails and the shared memory segment is not attached if one or more of the following are true:

EACCES Operation permission is denied to the calling process.

ENOMEM The available data space in memory is not large enough to hold the shared memory segment.

EINVAL The **shmid** parameter is not a valid shared memory identifier.

EINVAL The **shmaddr** parameter is not equal to 0, and the value of (**shmaddr** - (**shmaddr** module **SHMLBA**)) is an illegal address.

EINVAL The **shmaddr** parameter is not equal to 0, **SHM_RND** is not set in **shmflg**, and the the **shmaddr** parameter is an illegal address.

EINVAL The **shmaddr** parameter is not equal to 0, **SHM_RND** is not set in **shmflg**, and the the **shmaddr** parameter does not point to a segment boundary.

EMFILE The number of shared memory segments attached to the calling process would exceed the system-imposed limit.

Related Information
In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, "exit, _exit" in topic 1.2.73, " fork, vfork" in topic 1.2.83, "shmctl" in topic 1.2.259, "shmdt" in topic 1.2.260, and "shmget" in topic 1.2.261.

1.2.259 shmctl

Purpose
Controls shared memory operations.

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (shmid, cmd, buf)
int shmid, cmd;
struct shmid_ds *buf;
```

Description
The **shmctl** system call performs a variety of shared memory control operations as specified by the **cmd** parameter. The **shmid** parameter is a shared memory identifier returned by the **shmget** system call. The following **cmds** are available:

IPC_STAT Places the current value of each member of the data structure associated with the **shmid** parameter into the **shmid_ds** structure pointed to by the **buf** parameter. The current process must have read permission in order to perform this operation. The **shmid_ds** structure is defined in the **sys/shm.h** header file, and it contains the following members:

```
struct ipc_perm shm_perm;  /* Operation permission structure */
int     shm_segsz;         /* Segment size */
struct vseg *shm_vseg;     /* Segment identifier */
pid_t   shm_lpid;          /* ID of last process to call shmop */
pid_t   shm_cpid;          /* ID of process that created this shmid */
ushort  shm_nattch;        /* Current number of processes attached */
ushort  shm_cnattach;      /* No. of in-memory processes attached */
time_t  shm_atime;         /* Time of last shmat call */
time_t  shm_dtime;         /* Time of last shmdt call */
time_t  shm_ctime;         /* Time of the last change to this */
                           /*  structure with a shmctl call */
```

IPC_SET Sets the value of the following members of the data structure associated with the **shmid** parameter to the corresponding value found in the structure pointed to by the **buf** parameter:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode  /* Only the low-order nine bits */
```

This **cmd** can only be performed by a process that has an effective user ID equal to either that of superuser or to the value of **shm_perm.uid** in the data structure associated with the **shmid** parameter.

IPC_RMID Removes the shared memory identifier specified by the **shmid** parameter from the system and erases the shared memory segment and data structure associated with it. This **cmd** can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of **shm_perm.uid** in the data structure associated with the **shmid** parameter.

Note: In a Transparent Computing Facility cluster, shared memory segments exist only on the cluster site on which they are created. Consequently, processes that use shared memory segments cannot be migrated to other sites and processes on different cluster sites cannot share memory segments.

Return Value
Upon successful completion, a value of 0 is returned. If **shmctl** fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **shmctl** system call fails if one or more of the following are true:

EINVAL The **shmid** parameter is not a valid shared memory identifier.

EINVAL The **cmd** parameter is not a valid command.

EACCES The **cmd** parameter is equal to **IPC_STAT** and read permission is denied to the calling process.

EPERM The **cmd** parameter is equal to **IPC_RMID** or **IPC_SET**, and the effective user ID of the calling process is neither equal to the superuser ID, nor is it equal to the value of **shm_perm.uid** in the data structure associated with **shmid**.

EFAULT The **buf** parameter points to a location outside of the process's allocated address space.

Related Information
In this book: "disclaim" in topic 1.2.62, "shmat" in topic 1.2.258, "shmdt" in topic 1.2.260, "shmget" in topic 1.2.261, and "master" in topic 2.3.32.

1.2.260 shmdt

Purpose
Detaches a shared memory segment.

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt (shmaddr)
char *shmaddr;
```

Description
The **shmdt** system call detaches, from the calling process's data segment, the shared memory segment located at the address specified by the **shmaddr** parameter.

Return Value
Upon successful completion, a value of 0 is returned. If **shmdt** fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **shmdt** system call fails and the shared memory segment is not detached if the following is true:

EINVAL The **shmaddr** parameter is not the data segment start address of a shared memory segment.

ETXTBSY The **shmdt** system call attempted to detach a segment attached to a shared library.

Related Information
In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, "exit, _exit" in topic 1.2.73, " fork, vfork" in topic 1.2.83, "shmat" in topic 1.2.258, "shmctl" in topic 1.2.259, and "shmget" in topic 1.2.261.

1.2.261 shmget

Purpose
Gets shared memory segment.

Syntax

```
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key, size, shmflg)
key_t key;
int size, shmflg;
```

Description
The **shmget** system call returns the shared memory identifier associated with the specified **key**. The **key** parameter is either the value **IPC_PRIVATE** or an IPC key constructed by the **ftok** subroutine (or by a similar algorithm). See " stdipc: ftok" in topic 1.2.284 for details about this subroutine. The **size** parameter specifies the number of bytes of shared memory required.

The **shmflg** parameter is constructed by logically ORing one or more of the following values:

IPC_CREAT	Creates the data structure if it does not already exist.
IPC_EXCL	Causes the **shmget** system call to fail if **IPC_CREAT** is also set and the data structure already exists.
S_IRUSR	Permits the process that owns the data structure to read it.
S_IWUSR	Permits the process that owns the data structure to modify it.
S_IRGRP	Permits the group associated with the data structure to read it.
S_IWGRP	Permits the group associated with the data structure to modify it.
S_IROTH	Permits others to read the data structure.
S_IWOTH	Permits others to modify the data structure.

The values that begin with **S_I**- are defined in the **sys/stat.h** header file and are a subset of the access permissions that apply to files.

A shared memory identifier, its associated data structure, and a shared memory segment equal in bytes to the value of the **size** parameter are created for the **key** parameter if one of the following is true:

- The **key** parameter is equal to **IPC_PRIVATE**.
- The **key** parameter does not already have a shared memory identifier associated with it, and **IPC_CREAT** is set.

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

- **shm_perm.cuid** and **shm_perm.uid** are set equal to the effective user ID of the calling process.
- **shm_perm.cgid** and **shm_perm.gid** are set equal to the effective group ID of the calling process.

The low-order nine bits of **shm_perm.mode** are set equal to the low-order nine bits of the **shmflg** parameter.

shm_segsz is set equal to the value of the **size** parameter.

shm_lpid, **shm_nattch**, **shm_atime**, and **shm_dtime** are set equal to 0.

shm_ctime is set equal to the current time.

Note: In a Transparent Computing Facility cluster, shared memory segments exist only on the cluster site on which they are created. Consequently, processes that use shared memory segments cannot be migrated to other sites and processes on different cluster sites cannot share memory segments.

Return Value
Upon successful completion, a shared memory identifier is returned. If **shmget** fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **shmget** system call fails if one or more of the following are true:

EINVAL The **size** parameter is less than the system-imposed minimum or greater than the system-imposed maximum.

EACCES A shared memory identifier exists for the **key** parameter but operation permission as specified by the low-order nine bits of the **shmflg** parameter is not granted.

EINVAL A shared memory identifier exists for **key**, but the size of the segment associated with it is less than the **size** parameter and the **size** parameter is not equal to 0.

ENOENT A shared memory identifier does not exist for the **key** parameter and **IPC_CREAT** not set.

ENOSPC A shared memory identifier is to be created but the system-imposed limit on the maximum number of allowed shared memory identifiers system wide will be exceeded.

ENOMEM A shared memory identifier and associated shared memory segment are to be created but the amount of available physical memory is not sufficient to fill the request.

EEXIST A shared memory identifier exists for the **key** parameter, and both **IPC_CREAT** and **IPC_EXCL** are set.

Related Information
In this book: "shmat" in topic 1.2.258, "shmctl" in topic 1.2.259, "shmdt" in topic 1.2.260, and " stdipc: ftok" in topic 1.2.284.

1.2.262 shutdown

Purpose
Shuts down part or all of a full-duplex connection.

Syntax

```
int shutdown (s, how)
int s, how;
```

Description
The **shutdown** system call allows you to disable receives, sends, or both on the socket specified by the **s** parameter. The action of the system call is determined by the **how** parameter, according to the following values:

0 Disallows further receives.

1 Disallows further sends.

2 Disallows both further sends and receives.

Return Value

Upon successful completion, a value of 0 is returned. If the **shutdown** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The system call fails if one or more of the following are true:

EBADF The **s** parameter is not valid.

ENOTSOCK The **s** parameter refers to a file, not a socket.

ENOTCONN The socket is not connected.

EINVAL The *how* parameter is invalid.

Related Information
In this book: "connect" in topic 1.2.49 and "socket" in topic 1.2.275.

1.2.263 sigaction, sigvec, signal

Purpose
Specifies the action to take upon delivery of a signal.

Syntax

```
#include <signal.h>


int sigaction  (sig, act, oact)
int sig;
struct sigaction *act, *oact;
```

Description
The **sigaction** system call allows the calling process to examine and change the action to be taken when a specific signal is delivered to the process.

The **sig** parameter specifies the signal. If the **act** parameter is not NULL, it points to a **sigaction** structure that describes the action to be taken on receipt of the **sig** signal. If the **oact** parameter is not NULL, it points to a **sigaction** structure in which the signal action data in effect at the time of the **sigaction** call is returned. If the **act** parameter is NULL, signal handling is unchanged. Thus, the call can be used to inquire about the current handling of a given signal.

The **sigaction** structure has the following members:

```
void  (*sa_handler)();
sigset_t  sa_mask;
int       sa_flags;
```

The **sa_handler** field may have the value **SIG_DFL**, **SIG_IGN**, or may be a pointer to a function. A value of **SIG_DFL** requests default action to be taken when the signal is delivered. A value of **SIG_IGN** requests that the signal have no effect on the receiving process. A pointer to a function requests that the signal be caught (that is, the signal should cause the function to be called). These actions are more fully described below.

The **sa_mask** field can be used to specify that individual signals, in addition to those in the process's signal mask, are blocked from being delivered while the signal handler function specified in **sa_handler** is executing. See "sigemptyset, sigfillset, sigaddset, sigdelset, sigismember" in topic 1.2.264 for an explanation of the use of the **sa_mask** field. The **sa_flags** field may have the bits **SA_RESTART**, **SA_ONSTACK**, **SA_OLDSTYLE**, or **SA_NOCLDSTOP** set to specify further control over the actions taken upon delivery of a signal.

If the **SA_ONSTACK** bit is set, the system runs the signal-catching function on the signal stack specified by the **sigstack** system call. If this bit is not set, then the function executes on the stack of the process to which the signal is delivered.

If the **SA_OLDSTYLE** bit is set, the signal action is set to **SIG_DFL** (except for **SIGILL**, **SIGTRAP**, and **SIGPWR**) prior to calling the signal-catching function. This is supported for compatibility with old applications, and is not recommended since the same signal could reoccur before the signal-catching routine was able to reset the signal action. The default action (normally termination) would be taken in that case.

If a signal for which a signal-catching function exists is sent to a process while that process is executing certain system calls, the call terminates prematurely with a -1 return code and an **errno** value of EINTR. If the **SA_RESTART** bit is set in the **sa_flags** field of the handler for the signal, the system call does not fail but is restarted automatically. The affected system calls are **read** and **write** on a slow device (such as a terminal but not a regular file) and the **wait** system call.

The **sig** parameter can be any one of the following signal values except **SIGKILL**. Each of the names shown below is defined in the **signal.h** header file with the value of the corresponding signal number.

SIGHUP	1	Hangup
SIGINT	2	Interrupt
SIGQUIT	3*	Quit
SIGILL	4*	Illegal instruction
SIGTRAP	5*	Trace trap
SIGABRT	6*	Abort process (see "abort" in topic 1.2.7)
SIGEMT	7*	EMT instruction
SIGFPE	8*	Floating-point exception
SIGKILL	9	Kill (may not be caught, blocked, or ignored)
SIGBUS	10*	Bus error
SIGSEGV	11*	Segmentation violation
SIGSYS	12*	Bad parameter to system call
SIGPIPE	13	Write on a pipe with no one to read it
SIGALRM	14	Alarm clock
SIGTERM	15	Software termination signal
SIGURG	16+	Urgent condition on I/O channel
SIGSTOP	17@	Stop (may not be caught, blocked, or ignored)
SIGTSTP	18@	Interactive stop signal from TTY
SIGCONT	19!	Continue if stopped
SIGCHLD	20+	A child has stopped or exited
SIGTTIN	21@	Read of control TTY attempted from background
SIGTTOU	22@	Write to control TTY attempted from background
SIGIO	23+	Input/Output possible or complete
SIGXCPU	24	CPU time limit exceeded (see **setrlimit**)

SIGXFSZ	25	File size limit exceeded (Not sent by **write** in AIX)
reserved	26	
SIGMSG	27#	HFT input data pending
SIGWINCH	28+	Window size change
SIGPWR	29+	Power failure imminent (save your data)
SIGUSR1	30	User-defined signal 1
SIGUSR2	31	User-defined signal 2
SIGPROF	32	Profiling time alarm (see **setitimer**, "getitimer, setitimer" in topic 1.2.101)
SIGDANGER	33%+	System crash imminent
SIGVTALRM	34	Virtual time alarm (see **setitimer**, "getitimer, setitimer" in topic 1.2.101)
SIGMIGRATE	35$	Migrate the process to another CPU
SIGPRE	36*	Programming exception. On the PS/2, this is generated by the failure of the test performed by the 80386 BOUND instruction.
reserved	37-59	
SIGGRANT	60#	HFT monitor mode granted
SIGRETRACT	61#	HFT monitor mode retracted
SIGSOUND	62#	HFT sound sequence has completed
reserved	63	

The symbols in the preceding table have the following meaning:

* Default action includes creating a core dump file.

@ Default action is to stop the process.

! Default action is to restart or continue the process.

\+ Default action is to ignore these signals.

$ Default action is to migrate the process to another CPU.

% The cause is a shortage of paging space. See the **gpgscln** and **killem** stanzas in "master" in topic 2.3.32.

For more information on the use of these signals, see "hft" in topic 2.5.11.

The three types of actions that can be associated with a signal: **SIG_DFL**,

SIG_IGN, or a **pointer to a function** are described as follows:

SIG_DFL -- Default action: Signal-specific default action

Except for those signal numbers marked with a +, @, $, or !, the default action for a signal is to terminate the receiving process with all of the consequences described in the **_exit** system call. In addition, a **memory image** file will be created in the current directory of the receiving process if **sig** is one for which an asterisk appears in the preceding list and the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists in the current directory and is writable, or it can be created. If the file must be created, it will have the following properties:

- The access permission code 0666 (0x1B6), modified by the file creation mask (see "umask" in topic 1.2.314)
- A file owner ID that is the same as the effective user ID of the receiving process
- A file group ID that is the same as the effective group ID of the receiving process.

For signal numbers marked with a !, the default action is to restart the receiving process if it is stopped, or to continue execution of the receiving process.

For signal numbers marked with a @, the default action is to stop the execution of the receiving process temporarily. When a process stops, a **SIGCHLD** signal is sent to its parent process, unless the parent process has set the **SA_NOCLDSTOP** flag. While a process is stopped, any additional signals that are sent to the process will not be delivered until the process is continued. An exception to this is **SIGKILL**, which always terminates the receiving process. Another exception is **SIGCONT**, which always causes the receiving process to restart or continue execution even if blocked or ignored. A process whose parent has terminated shall be sent a **SIGKILL** signal if the **SIGTSTP**, **SIGTTIN**, or **SIGTTOU** signals are generated for that process.

For signal numbers marked with a +, the default action is to ignore the signal. In this case, delivery of the signal has no effect on the receiving process.

If a signal action is set to **SIG_DFL** while the signal is pending, the signal remains pending.

SIG_IGN -- Ignore signal.

Delivery of the signal will have no effect on the receiving process. If a signal action is set to **SIG_IGN** while the signal is pending, the pending signal will be discarded.

An exception to this is the **SIGCHLD** signal whose **SIG_DFL** action is to ignore the signal. If **SIGCHLD** is set to **SIG_IGN**, it means that the process does not want to receive the **SIGCHLD** signal when one of its child processes dies, and does not want to have its **wait** calls return

because a child process is dead (just when no more child processes exist, and on stopped child processes).

Note: The **SIGKILL** and **SIGSTOP** signals cannot be ignored.

pointer to a function -- Catch signal.

Upon delivery of the signal, the receiving process is to execute the signal-catching function specified by the pointer to function. The signal-handler subroutine can be declared as follows:

```
handler (sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

The **sig** parameter is the signal number. The **code** parameter gives extra information about the cause of certain signals. For **SIGFPE**, **code** specifies the nature of the floating-point exception. For **SIGMIGRATE**, **code** is the number of the site to which the process should migrate. For other signals, **code** is always 0. The **scp** parameter points to the **sigcontext** structure that is later used to restore the process's previous execution context. The **sigcontext** structure is defined in **signal.h**.

A new signal mask is calculated and installed for the duration of the signal-catching function (or until **sigprocmask** or **sigsuspend** system calls are made). This mask is formed by taking the union of the process's signal mask, the mask associated with the action for the signal being delivered, and a mask corresponding to the signal being delivered. The mask associated with the signal-catching function is not allowed to block those signals that cannot be ignored. This is enforced by the kernel without causing an error to be indicated. If and when the signal-catching function returns, the original signal mask is restored (modified by any **sigprocmask** calls that were made since the signal-catching function was called) and the receiving process resumes execution at the point it was interrupted.

The signal-catching function can cause the process to resume in a different context by calling the **longjmp** subroutine. When the **longjmp** subroutine is called, the process's signal mask and signal stack state (stack pointer and on-signal-stack state) are restored to those in effect at the time the corresponding **setjmp** call was made.

Once an action is installed for a specific signal, it remains installed until another action is explicitly requested (by another call to the **sigaction** system call), or until one of the **exec** functions is called. An exception to this is when the **SA_OLDSTYLE** is set in which case the action of a caught signal gets set to **SIG_DFL**, except for **SIGILL**, **SIGTRAP**, and **SIGPWR**, prior to calling the signal-catching function for that signal.

If a signal action is set to a pointer to a function while the signal is pending, the signal will remain pending.

When signal-catching functions are invoked asynchronously with process execution, the behavior of some of the functions defined by this reference is unspecified if they are called from a signal-catching function. The following table defines a set of functions that shall be reentrant with respect to signals (that is, applications may invoke

them, without restriction, from signal-catching functions): **_exit, access, alarm, chdir, chmod, chown, close, creat, dup2, dup, exec, fcntl, fork, fstat, getegid, geteuid, getgid, getgroups, getpgrp, getpid, getppid, getuid, kill, link, lseek, mkdir, mkfifo, open, pause, pipe, read, rename, rmdir, setgid, setpgrp, setuid, sigaction, sigaddset, sigdelset, sigfillset, sigemptyset, sigismember, signal, sigpending, sigprocmask, sigsuspend, sleep, stat, time, times, umask, uname, unlink, ustat, utime, wait3, wait, write**. No other library functions should be called from signal-catching functions, since their behavior is undefined.

Note: The **SIGKILL** and **SIGSTOP** signals cannot be caught.

Compatibility Interfaces

```
#include <sys/signal.h>

int sigvec (sig, invec, outvec)
int sig;
struct sigvec *invec, *outvec;
```

The **sigvec** function is the same as a **sigaction** system call except that the **sigvec** structure is used instead of the **sigaction** structure. The **sigvec** structure specifies a mask as an **int** instead of a **sigset_t**. The mask for **sigvec** is constructed by setting the **i**-th bit in the mask if signal **i** is to be blocked. Therefore, **sigvec** only allows signals of value 1-32 to be blocked when a signal-handling function is called. The other signals will not be blocked by the signal-handler mask.

Note: The **SA_RESTART** flag cannot be specified with **sigvec**. Instead, for 4.3BSD compatibility, the flag **SV_INTERRUPT** can be specified. **SV_INTERRUPT** has the opposite effect of **SA_RESTART**.

```
#include <sys/signal.h> or <signal.h>

void (*signal (sig, action)) ( )
int sig;
void (*action) ( );
```

The **signal** function allows the action associated with a signal. The **action** parameter can have the same values that are described for the **sa_handler** field in the **act** structure of the **sigaction** system call. However, no signal handler mask or flags can be specified; the **signal** function implicitly sets the signal handler mask to not block the signal **sig** and the flags to be **SA_OLDSTYLE**. Also, a call to the function **signal** cancels a pending signal **sig**, except for a pending **SIGKILL**.

Upon successful completion of a **signal** call, the value of the previous signal action is returned. If the call fails, a value of **SIG_ERR** is returned and **errno** is set to indicate the error as in the **sigaction** call.

Note that **sigvec** and **signal** do not check for valid pointers, and therefore will not generate EFAULT.

Return Value

Upon successful completion, a value of 0 is returned. If the **sigaction** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions

The **sigaction** system call fails and no new signal handler is installed if one of the following occurs:

EFAULT The **act** or **oact** parameter points to a location outside of the process's allocated address space.

EINVAL The **sig** parameter is not a valid signal number.

EINVAL An attempt was made to ignore or supply a handler for **SIGKILL**, or **SIGSTOP**.

Related Information

In this book: "acct" in topic 1.2.11, "exit, _exit" in topic 1.2.73, "kill, kill3, killpg" in topic 1.2.138, "pause" in topic 1.2.202, "ptrace" in topic 1.2.212, "setjmp, longjmp, _setjmp, _longjmp" in topic 1.2.250, "sigprocmask, sigsetmask, sigblock" in topic 1.2.267, "sigemptyset, sigfillset, sigaddset, sigdelset, sigismember" in topic 1.2.264, "sigstack" in topic 1.2.268, "sigsuspend, sigpause" in topic 1.2.269, "umask" in topic 1.2.314, "wait, waitpid" in topic 1.2.325, and "core" in topic 2.3.10.

The **kill** command in *AIX Operating System Commands Reference*.

1.2.264 sigemptyset, sigfillset, sigaddset, sigdelset, sigismember

Purpose
Creates and manipulates signal masks.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <signal.h>

int sigemptyset (set)
sigset_t *set;

int sigfillset (set)
sigset_t *set;

int sigaddset (set, signo)
sigset_t *set;
int signo;

int sigdelset (set, signo)
sigset_t *set;
int signo;

int sigismember (set, signo)
sigset_t *set;
int signo;
```

Description
The **sigemptyset**, **sigfillset**, **sigaddset**, **sigdelset** and **sigismember** subroutines manipulate signal masks. These functions operate on data objects addressable by the application, not on any set of signals known to the system, such as the set blocked from delivery to a process or the set pending for a process (see "sigaction, sigvec, signal" in topic 1.2.263).

The **sigemptyset** function initializes the signal set pointed to by the parameter **set** such that all signals are excluded. The **sigfillset** function initializes the signal set pointed to by the parameter **set** such that all signals are included. A call to either **sigemptyset** or **sigfillset** must be made a least once for each object of type **sigset_t** prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of **sigaddset**, **sigdelset**, **sigismember**, **sigaction**, **sigprocmask**, or **sigsuspend**, the results are undefined.

The **sigaddset** and **sigdelset** functions respectively add and delete the individual signal specified by the **signo** parameter from the signal set specified by the **set** parameter. The **sigismember** function tests whether the **signo** is a member of the signal set pointed to by the **set** parameter.

Return Value
Upon successful completion, the **sigismember** function returns a value of one if the specified signal is a member of the specified set, or the value of 0 if not. Upon successful completion, the other functions return a value of 0. For all the above functions, if an error is detected, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **sigaddset** and **sigdelset** subroutines fail if the following is true:

EINVAL The value of the **signo** parameter is not a valid signal number, or **set** is NULL.

Example

To generate and use a signal mask that blocks only **SIGINT** from delivery:

```
#include <signal.h>
#include <unistd.h>

int return_value;
sigset_t newset;
sigset_t *newset_p;
...
newset_p = &newset;
sigemptyset(newset_p);
sigaddset(newset_p, SIGINT);
return_value = sigprocmask (SIG_SETMASK, newset_p, NULL);
```

Related Information

In this book: "sigprocmask, sigsetmask, sigblock" in topic 1.2.267, "sigsuspend, sigpause" in topic 1.2.269, and "sigaction, sigvec, signal" in topic 1.2.263.

1.2.265 siginterrupt

Purpose
Allows signals to interrupt system calls.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/signal.h>

int siginterrupt (sig, flag)
int sig, flag;
```

Description
The **siginterrupt** routine is used to change the restart behavior when a system call is interrupted by the specified signal. The restart behavior is set by the sigaction system call when a signal handler is installed. It can also be modified using the **sigaction** system call. The **siginterrupt** routine is provided only for compatibility with 4.3BSD systems.

If the flag is 0, then system calls will be restarted if they are interrupted by the specified signal and no data has been transferred yet.

If the flag is 1, then restarting of system calls is disabled. If a system call is interrupted by the specified signal and no data has been transferred, the system call will return -1 and set **errno** to **EINTR**. Interrupted system calls that have started transferring data will return the amount of data actually transferred.

Return Value
Upon successful completion, **siginterrupt** returns a value of 0. Otherwise, -1 is returned and **errno** is set to indicate the error.

Error Conditions

The **siginterrupt** routine fails if the following is true:

EINVAL The *sig* parameter is not a valid signal number.

Related Information
In this book: "sigaction, sigvec, signal" in topic 1.2.263, "sigprocmask, sigsetmask, sigblock" in topic 1.2.267.

1.2.266 sigpending

Purpose
Examines pending signals.

Syntax

```
#include <signal.h>

int sigpending (set)
sigset_t *set;
```

Description
The **sigpending** function will store the set of signals that are blocked from delivery and pending for the calling process, in the space pointed to by the argument **set**.

Return Value
Upon successful completion the value of 0 is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
Possible error conditions include:

EFAULT A **set** parameter points to a location outside the process's allocated address space.

Related Information
In this book: "sigprocmask, sigsetmask, sigblock" in topic 1.2.267.

1.2.267 sigprocmask, sigsetmask, sigblock

Purpose
Sets the current signal mask.

Syntax

```
int sigprocmask (how, set, oset)
int how;
sigset_t *set, *oset;
```

Description
The **sigprocmask** system call is used to examine and change the calling process's signal mask. If the value of the argument **set** is not NULL, it points to a signal mask to be used to change the current signal mask.

The **how** parameter indicates the manner in which the mask is changed. It may have one of the following values:

SIG_BLOCK The resulting mask is the union of the current mask and the signal mask pointed to by the **set** parameter.

SIG_UNBLOCK The resulting mask is the intersection of the current mask and the complement of the signal mask pointed to by the **set** parameter.

SIG_SETMASK The resulting mask is the signal mask pointed to by the **set** parameter.

If the **oset** parameter is not NULL, the signal mask in effect at the time of the call is stored in the space pointed to by the **oset** parameter. If the value of the **set** parameter is NULL, the value of the **how** parameter is not significant and the process's signal mask is unchanged. Thus, the call can be used to inquire about currently blocked signals.

Typically, you would use the **sigprocmask(SIG_BLOCK,...)** system call to block signals during a critical section of code and then use the **sigprocmask(SIG_SETMASK,...)** system call to restore the mask to the previous value returned by the **sigprocmask(SIG_BLOCK,...)** system call.

If there are any pending unblocked signals after the call to the **sigprocmask** system call, at least one of those signals will be delivered before the **sigprocmask** function returns.

The **sigprocmask** system call does not allow the **SIGKILL** or **SIGSTOP** signals to be blocked. If a program attempts to block one of these signals, **sigprocmask** gives no indication of the error.

Compatibility Interfaces

```
int sigsetmask (sigmask)
int sigmask;
```

The **sigsetmask** subroutine allows the changing the process's signal mask for signal values 1-32. This same function can be accomplished for all signal values with the **sigprocmask(SIG_SETMASK**,...,...) system call. The signal of value **i** will be blocked if the **i**-th bit of **sigmask** parameter is set.

Upon successful completion, the **sigsetmask** subroutine returns the value of

the previous signal mask. If the subroutine fails, a -1 is returned and **errno** is set to indicate the error as in the **sigprocmask** system call.

int sigblock (sigmask)
int sigmask;

The **sigblock** subroutine allows signals with values 1-32 to be ORed into the current process signal mask. This same function can be accomplished for all signal values with the **sigprocmask(SIG_BLOCK**,...,...) system call. In addition to those currently blocked, the signal of value **i** is blocked if the **i**-th bit of **sigmask** parameter is set.

Upon successful completion, the **sigblock** subroutine returns the value of the previous signal mask. If the subroutine fails, a -1 is returned and **errno** is set to indicate the error as in the **sigprocmask** system call.

Return Value
Upon successful completion, a value of 0 is returned. If the **sigprocmask** system call fails, the process's signal mask is unchanged, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **sigprocmask** system call fails if one of the following is true:

EINVAL The value of the **how** parameter is not equal to one of the defined values.

EFAULT The **set** or **oset** parameter points to a location outside the process's address space.

Example
To set the signal mask to block only **SIGINT** from delivery:

```
#include <signal.h>
#include <unistd.h>

int return_value;
sigset_t newset;
sigset_t *newset_p;
...
newset_p = &newset;
sigemptyset(newset_p);
sigaddset(newset_p, SIGINT);
return_value = sigprocmask (SIG_SETMASK, newset_p, NULL);
```

Related Information
In this book: "kill, kill3, killpg" in topic 1.2.138, "sigaction, sigvec, signal" in topic 1.2.263, and "sigsuspend, sigpause" in topic 1.2.269.

1.2.268 sigstack

Purpose
Sets and gets signal stack context.

Syntax

#include <signal.h>

int sigstack (instack, outstack)
struct sigstack *instack, *outstack;

Description
The **sigstack** system call defines an alternate stack on which signals are to be processed.

If the value of the **instack** parameter is nonzero, it points to a **sigstack** structure, which has the following members:

```
char     *ss_sp;
int      ss_onstack;
```

The value of **instack->ss_sp** specifies the stack pointer of the new signal stack. Since stacks grow from numerically greater addresses to lower ones, the stack pointer passed to the **sigstack** system call should point to the numerically high end of the stack area to be used.
instack->ss_onstack should be set to 1 if the process is currently executing on that stack; otherwise, it should be 0.

If the value of the **outstack** parameter is nonzero, it points to a **sigstack** structure into which the **sigstack** system call stores the current signal stack state.

If the value of the **instack** parameter is NULL, the signal stack state is not set. If the value of the **outstack** parameter is NULL, the previous signal stack state is not reported.

When a signal whose handler is to run on the signal stack occurs, the system checks to see if the process is already executing on that stack. If so, it continues to do so even after the handler returns. If not, the signal handler runs on the signal stack, and the original stack is restored when the handler returns.

Use the **sigaction** system call to specify whether a given signal's handler routine should run on the signal stack.

Warning: A signal stack does not automatically increase in size as a normal stack does. If the stack overflows, unpredictable results may occur.

Return Value
Upon successful completion, a value of 0 is returned. If the **sigstack** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **sigstack** system call fails and the signal stack context remains unchanged if the following is true:

EFAULT The **instack** or **outstack** parameter points to a location outside

of the process's allocated address space.

Related Information

In this book: "setjmp, longjmp, _setjmp, _longjmp" in topic 1.2.250, and "sigaction, sigvec, signal" in topic 1.2.263.

1.2.269 sigsuspend, sigpause

Purpose
Atomically changes the set of blocked signals and waits for an interrupt.

Syntax

```
#include <signal.h>

int sigsuspend (sigmask)
sigset_t *sigmask;
```

Description
The **sigsuspend** system call replaces the process's signal mask with the signal mask pointed to by the **sigmask** parameter. It then suspends execution of the process until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. The **sigsuspend** system call does not allow the **SIGKILL**, **SIGSTOP**, or **SIGCONT** signals to be blocked. If a program attempts to block one of these signals, **sigsuspend** gives no indication of the error.

If delivery of a signal causes the process to terminate, the **sigsuspend** system call does not return. If delivery of a signal causes a signal-catching function to execute, the **sigsuspend** system call returns after the signal-catching function returns, with the signal mask restored to the set that existed prior to the **sigsuspend** call.

The **sigsuspend** system call sets the signal mask and waits for an unblocked signal as one ***atomic operation***. This means that signals cannot occur between the operations of setting the mask and waiting for a signal. If a program invokes the **sigprocmask(SIG_SETMASK,...)** and **pause** system calls separately, a signal that occurs between these system calls might not be noticed by **pause**.

In normal usage, a signal is blocked by using the **sigprocmask(SIG_BLOCK,...)** system call at the beginning of a critical section. The process then determines whether there is work for it to do. If there is no work, the process waits for work by calling **sigsuspend** with the mask previously returned by the **sigprocmask** system call.

Compatibility Interfaces

```
int sigpause (sigmask)
int sigmask;
```

The **sigpause** function only allows signals with values 1-32 to be masked. Signal **i** is blocked if the **i**-th bit in the **sigmask** parameter is a 1.

Return Value
If a signal is caught by the calling process and control is returned from the signal handler, the calling process resumes execution after the **sigsuspend** system call, which always returns a value of -1 and sets **errno** to EINTR.

Related Information
In this book: "pause" in topic 1.2.202, "sigaction, sigvec, signal" in topic 1.2.263, and "sigprocmask, sigsetmask, sigblock" in topic 1.2.267.

1.2.270 sin, cos, tan, asin, acos, atan, atan2

Purpose
Computes trigonometric functions.

Library
Math Library (**libm.a**)

Syntax

```
#include <math.h>

double sin (x)                  double asin (x)
double x;                       double x;
double cos (x)                  double acos (x)
double x;                       double x;
double tan (x)                  double atan (x)
double x;                       double x;
                                double atan2 (y, x)
                                double x, y;
```

Description
The **sin**, **cos**, and **tan** subroutines return the sine, cosine and tangent, respectively, of their parameters, which are in radians.

The **asin** subroutine returns the arcsine of **x**, in the range -&pi./2 to &pi./2.

The **acos** subroutine returns the arccosine of **x**, in the range 0 to &pi..

The **atan** subroutine returns the arctangent of **x**, in the range -&pi./2 to &pi./2.

The **atan2** subroutine returns the arctangent of **y/x**, in the range -&pi. to &pi., using the signs of both parameters to determine the quadrant of the return value.

Error Conditions

These subroutines can perform either of the following types of error handling. Both types of error handling allows you to define special actions to be taken when an error occurs.

On the PS/2 only, exception handling is performed by default accordin to ANSI/IEEE standard 754 for binary floating-point arithmetic for arguments **x** in the range between -2(63) and 2(63). Otherwise, the behavior of these subroutines is undefined.

If a hardware floating point processor is installed in your system, then using this option can provide greater performance in addition to IEEE exception handling. This mode instructs the C compiler to generate code that avoids the overhead of the math library subroutines by generating math coprocessor code in-line.

On the AIX/370, **matherr** error handling is performed by default (see **matherr** handling, as described on page 1.2.163). To activate **matherr** error handling on the PS/2, include the **-z** option on the **cc** command line when compiling source code. The default error-handling

procedures for these subroutines are as follows:

sin, **cos**, **tan**
The **sin**, **cos** and **tan** subroutines lose accuracy when passed a large value for the **x** parameter. For sufficiently large parameters, these functions return 0 when there would otherwise be a complete loss of significance. In this case, a message that indicates a TLOSS error is written to standard error. For less extreme values, a PLOSS error is generated but no message is written. In both cases, **errno** is set to ERANGE.

The **tan** subroutine can return ±HUGE if its parameter is near an odd multiple of &pi./2 when the correct value would overflow, and sets **errno** to ERANGE.

asin, **acos**
The **asin** and **acos** subroutines return 0 and set **errno** to EDOM if their parameters are larger than 1.0. In addition, an error message that indicates a domain error is written to the standard error output.

Related Information
In this book: "matherr" in topic 1.2.163.

1.2.271 sinh, cosh, tanh

Purpose
Computes hyperbolic functions.

Library
Math Library (**libm.a**)

Syntax

#include <math.h>

```
double sinh (x)
double x;
double cosh (x)
double x;
```

```
double tanh (x)
double x;
```

Description
The **sinh** subroutine returns the hyperbolic sine of its parameter. The **cosh** subroutine returns the hyperbolic cosine of its parameter. The **tanh** subroutine returns the hyperbolic tangent of its parameter.

The **sinh** and the **cosh** subroutines return HUGE if the correct value overflows. **errno** is also set to ERANGE.

You can use the **matherr** subroutine to change these error-handling procedures. See "matherr" in topic 1.2.163 for details.

Error Conditions
The **sinh**, **cosh**, and **tanh** subroutines fail if the following is true:

EDOM The value of **x** is NaN.

The **cosh** and **sinh** subroutines fail if the following is true:

ERANGE The result would cause overflow.

1.2.272 site

Purpose
Returns the site number on which the specified process is running.

Syntax

#include <sys/types.h>

siteno_t site(pid)
pid_t pid;

Description
The **site** system call returns the site number of the specified process. If **pid** is 0, the current process is assumed. If the specified process cannot be located, -1 is returned.

Return Value
Upon successful completion, a nonnegative value for **siteno_t** is returned indicating the site number. Otherwise, a -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **site** system call fails if the following is true:

ESRCH The process ID specified is invalid.

Examples

```
#include <sys/types.h>

/* This program prints the number of the site where it is executed */
main()
{
 printf(''mysite is %d\n'', site((pid_t) 0));
}
```

Related Information
In this book: "sfent, sfnum, sfname, sfctype, sfxcode, setsf, endsf" in topic 1.2.257.

1.2.273 sleep

Purpose
Suspends execution of the current process for an interval of time.

Library
Standard C Library (**libc.a**)

Syntax

unsigned int sleep (seconds)
unsigned int seconds;

Description
The **sleep** subroutine causes the current process to suspend execution for the number of seconds specified by the **seconds** parameter. The sleep routine sets an alarm and pauses until that alarm or some other signal occurs.

The actual sleep time of the process may be either shorter or longer than the requested sleep time. The sleep time may be shorter because:

Wakeups occur on the second at fixed one-second intervals according t an internal clock.

Any caught signal terminates the sleep following execution of tha signal's catching routine.

The sleep time may be longer than the requested sleep time due to the scheduling of other activities in the system.

The value returned by the **sleep** subroutine is the requested sleep time minus the time actually slept.

The process calling the **sleep** subroutine may set an alarm prior to calling the **sleep** subroutine.

If a previous alarm has been set, and the **sleep** subroutine's sleep time exceeds the process's previously set alarm time, the process only sleeps until the time specified by the previously set alarm and the calling process's alarm catch routine is executed just before the **sleep** subroutine returns.

If a previous alarm has been set, and the **sleep** subroutine's sleep time is less than the process's previously set alarm time, the current process is suspended from execution for the number of seconds specified by the **sleep** subroutine. The previously set alarm is reset to go off at the same time it would have without the **sleep** subroutines intervention.

Warning: The results are undefined if, while it is sleeping, the calling program issues any other **alarm** or **sleep** calls. This can happen if a signal arrives in the interim and the signal handler calls **alarm** or **sleep**.

Error Conditions
The **sleep** subroutine is always successful and no return value is reserved to indicate an error.

Related Information
In this book: "alarm" in topic 1.2.14, "pause" in topic 1.2.202, and "sigaction, sigvec, signal" in topic 1.2.263.

1.2.274 snap

Purpose
Provide user access to perform dumps.

Library
Standard C Library (**libc.a**)

Syntax

```
int snap (a_flag)
int a_flag;
```

Description
The **snap** system call provides a user system call interface to perform a system dump. This is a memory dump on a System/370 only, and may or may not be performed at panic time.

The **a_flag** parameter specifies whether or not the system should panic when the memory dump is taken. If **a_flag** is non-zero, the system will perform a panic and then dump memory. You must be superuser to perform this system call.

Return Value
If the **snap** system call is unsuccessful, -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **snap** system call fails if the following is true:

EPERM Operation not permitted; permission is denied.

1.2.275 socket

Purpose
Creates an endpoint for communication and returns a descriptor.

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>

int socket  (domain, type, protocol)
int domain, type, protocol;
```

Description
The **socket** system call creates an endpoint for communication and returns a socket descriptor.

The **domain** parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. The protocol family generally is the same as the address format for the addresses supplied in later operations on the socket. These formats are defined in the **sys/socket.h** header file. The formats are:

AF_UNIX AIX path names
AF_INET ARPA Internet addresses.

The value of the **type** parameter specifies the semantics of communication. AIX supports these types:

SOCK_STREAM Provides sequenced, two-way byte streams with a transmission mechanism for out-of-band data.
SOCK_DGRAM Provides ***datagrams***, which are connectionless messages of a fixed maximum length (usually small).

The **protocol** parameter specifies a particular protocol to be used with the socket. In most cases, a single protocol exists to support a particular socket type using a given address format. When many protocols exist, you must specify a particular protocol. Use the number for the communication domain in which the communication takes place.

The different types of sockets available are used for different purposes. **SOCK_DGRAM** sockets allow sending datagrams to correspondents named in **send** socket calls. Programs can also receive datagrams via sockets by using the **recv** system call.

SOCK_STREAM sockets are full-duplex byte streams. A stream socket must be connected before any data may be sent or received on it. Create a connection to another socket with the **connect** system call. Once connected, use the **read** and **write** system calls, or the **send** and **recv** system calls to transfer data. Issue the **close** system call when a session is finished. Use the **send** and **recv** system calls for out-of-band data.

SOCK_STREAM communications protocols are designed to prevent the loss or duplication of data. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable period of time, the connection is broken. When this occurs, the socket system calls indicate an error with a return value of -1 and with **ETIMEDOUT** as the specific code written to the global variable **errno**. If a process sends on a broken stream, a **SIGPIPE** signal is raised. Processes that

cannot handle the signal terminate.

An **fcntl** system call can be used to specify a process group to receive a **SIGURG** signal when out-of-band data arrives on a socket. It may also be used to enable non-blocking I/O and asynchronous notification of I/O events via the **SIGIO** signal.

The operation of sockets is controlled by socket level options. The **getsockopt** and **setsockopt** system calls are used to get and set these options, which are defined in the **sys/socket.h** file. See "getsockopt, setsockopt" in topic 1.2.121 for information on how to use these options.

Return Value

Upon successful completion, a descriptor referring to the socket is returned. If the **socket** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions

The system call fails if one or more of the following are true:

EPROTONOSUPPORT
: The protocol type or the specified protocol is not supported within this domain.

EMFILE
: The **per-process** descriptor table is full.

ENOBUFS
: Insufficient resources were available in the system to complete the call.

ENFILE
: The system file table is full.

EACCES
: Permission to create a socket of the specified type and/or protocol is denied.

Related Information

In this book: "accept" in topic 1.2.9, "bind" in topic 1.2.20, "getsockname" in topic 1.2.120, "getsockopt, setsockopt" in topic 1.2.121, "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "listen" in topic 1.2.157, "recv, recvfrom, recvmsg" in topic 1.2.227, "select" in topic 1.2.242, "send, sendto, sendmsg" in topic 1.2.246, "shutdown" in topic 1.2.262, "connect" in topic 1.2.49, and "socketpair" in topic 1.2.276.

1.2.276 socketpair

Purpose
Creates a pair of connected sockets.

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>

socketpair (d, type, protocol, sv)
int d, type, protocol;
int sv[2];
```

Description
The **socketpair** system call creates an unnamed pair of connected sockets in the specified domain **d**, of the specified **type**, and using the optionally specified **protocol**. The descriptors used in referencing the new sockets are returned in **sv**[0] and **sv**[1]. The two sockets are identical.

Return Value
Upon successful completion, a value of 0 is returned. If the **socketpair** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The system call fails if one or more of the following are true:

EMFILE This process has too many descriptors in use.

EPROTONOSUPPORT
The specified protocol or address family cannot be used on this system.

EOPNOSUPPORT The specified protocol does not allow create of socket pairs.

EFAULT The **sv** parameter is not in a writable part of the user address space.

Related Information
In this book: "socket" in topic 1.2.275.

1.2.277 sockets library

Purpose
Provides communications between processes.

Library
Internet Library (**libc.a**)

Description
This section contains a list of the socket routines provided by AIX, an overview of sockets and how to use them, and a list of related publications on networks and communications that you may find useful.

Subtopics
1.2.277.1 Socket Routines
1.2.277.2 Overview of Sockets
1.2.277.3 Socket Names
1.2.277.4 Related Network Publications

1.2.277.1 Socket Routines

The following table is an list of the socket routines and a brief description of each.

accept	Accepts a connection on a socket.
bind	Binds a name to a socket.
connect	Initiates a connection on a socket.
gethostbyaddr	Gets network host address.
gethostbyname	Gets network host name.
sethostent	Opens and rewinds the host entry file.
endhostent	Closes the host entry file.
gethostid	Gets the unique identifier of the current host.
sethostid	Sets the unique identifier of the current host.
gethostname	Gets the name of the current host.
sethostname	Sets the name of the current host.
getnetent	Gets network entry.
getnetbyaddr	Gets network entry by address.
getnetbyname	Gets network entry by name.
setnetent	Opens and rewinds the network entry file.
endnetent	Closes the network entry file.
getpeername	Gets the name of the connected peer.
getprotoent	Gets protocol entry.
getprotobynumber	Gets protocol entry by number.
getprotobyname	Gets protocol entry by name.
setprotoent	Opens and rewinds the protocol entry file.
endprotoent	Closes the protocol entry file.
getservent	Gets service entry.
getservbyname	Gets service entry by name.
getservbyport	Gets service entry by port number.
setservent	Opens and rewinds the service entry file.
endservent	Closes the service entry file.
getsockname	Gets the socket name.

getsockopt	Gets options on sockets.
setsockopt	Sets options on sockets.
htonl, htons	Converts values between host and Internet network byte order.
ntohl, ntohs	Converts values between Internet network and host byte order.
inet_addr	Returns a string representing an Internet address.
inet_network	Returns a string representing an Internet network number.
inet_ntoa	Converts Internet address into an ASCII string.
inet_makeaddr	Constructs an Internet address from a Internet network number and a local network address.
inet_lnaof	Returns the local network address number from an Internet address.
inet_netof	Returns the network number from an Internet address.
listen	Listens for connections on a socket.
rcmd	Allows execution of commands on a remote host.
recv	Receives a message from a connected socket only.
recvfrom	Receives a message from a socket.
recvmsg	Receives a message from a socket.
resolver	A set of functions that resolves domain names. Contains the following subroutines: **res_mkquery** **res_send** **res_init** **dn_comp** **dn_expand** **getshort** **putshort** **putlong**
rexec	Allows command execution on a remote host.
rresvport	Obtains a socket with a privileged address bound to it.
ruserok	Provides authentication of remote requests.
send	Sends a message from a socket only when socket is in connected state.
sendto	Sends a message from a socket.
sendmsg	Sends a message from a socket.

shutdown	Shuts down part or all of a full-duplex connection.
socket	Creates an endpoint for communications and returns a descriptor.
socketpair	Creates a pair of connected sockets.

1.2.277.2 Overview of Sockets

A ***socket*** is an object that provides communications between processes. Sockets are referenced by file descriptors and have qualities similar to those of a character special device. Read, write, and select operations can be performed on sockets by using the appropriate system calls.

A socket is created with the **socket** system call. (See "socket" in topic 1.2.275.) This system call creates a socket of a specified domain, type, and protocol. Sockets have different qualities depending on these specifications.

A ***domain*** is a name space or an address space. Each domain has different rules for valid names and interpretation of names. After a socket is created, it can be given a name, according to the rules of the domain in which it was created.

AIX provides support for the following socket domains:

Local Provides socket communication between processes running on the same AIX system when a domain of **AF_UNIX** is specified. A name in this domain is a string of ASCII characters whose maximum length is machine dependent.

Internet Provides socket communication between a local process and a process running on a remote host when a domain of **AF_INET** is specified. This domain requires that IBM AIX TCP/IP be installed on your system. A name in this domain is a DARPA Internet address, made up of a 32-bit IP address and a 16-bit port address. (See the discussion of addresses and names in *AIX TCP/IP User's Guide*.)

In AIX, there are two types of sockets:

SOCK_DGRAM Provides datagrams, which are connectionless messages of a fixed maximum length. This type of socket is generally used for short messages, such as a name server or time server, since the order and reliability of message delivery is not guaranteed.

In the local domain, **SOCK_DGRAM** is similar to a message queue. In the Internet domain, **SOCK_DGRAM** is implemented on the UDP/IP protocol.

SOCK_STREAM Provides sequenced, two-way byte streams with a transmission mechanism for out-of-band data. The data is transmitted on a reliable basis, in order.

In the local domain, **SOCK_STREAM** is like a pipe. In the Internet domain, **SOCK_STREAM** is implemented on the TCP/IP protocol.

A ***protocol*** is a standard set of rules for transferring data, such as UDP/IP and TCP/IP. A protocol is specified only if more than one protocol is supported for this particular socket type in this domain. Otherwise, this parameter is set to 0.

1.2.277.3 Socket Names

A socket name, which is also called a socket address, is specified by the **sockaddr** structure. This structure is defined in the **sys/socket.h** header file, and it contains the following members:

```
ushort  sa_family;    /* Defines socket address family */
char    sa_data[14];  /* Contains up to 14 bytes of direct address */
```

The **sa_family** is the address family or domain, either **AF_UNIX** for the local domain or **AF_INET** for the Internet domain. The contents of **sa_data** depend on the protocol in use, but generally a socket name consists of a machine name part and a port or service name part.

1.2.277.4 Related Network Publications

For general information about networking, the following publications are recommended. These publications are distributed by the Network Information Center on behalf of the Defense Communications Agency and Defense Advanced Research Projects Agency (DARPA). The mailing address is:

Network Information Center
SRI International
Menlo Park, CA 92025

Assigned Numbers, RFC990, J. Reynolds, J. Postel

Broadcasting Internet Datagrams, RFC919, J. Mogul

Domain Names - Concepts and Facilities, RFC882, P. Mockapetris

Domain Names - Implementation and Specification, RFC883, P. Mockapetris

File Transfer Protocol, RFC959, J. Postel

Internet Control Message Protocol, RFC792, J. Postel

Internet Name Server Protocol, IEN116, J. Postel

Internet Protocol, RFC791, J. Postel

Internet Standard Subnetting Procedure, RFC950, J. Mogul

Name/Finger, RFC742, K. Harrenstien

Official ARPA-Internet Protocols, RFC944, J. Reynolds, J. Postel

Simple Mail Transfer Protocol, RFC821, J. Postel

Standard for the Format of ARPA Internet Text Messages, RFC822.

Telnet Binary Transmission, RFC856, J. Postel, J. Reynolds

Telnet Option Specifications, RFC855, J. Postel, J. Reynolds

Telnet Protocol Specification, RFC854, J. Postel, J. Reynolds

Telnet Terminal Type Option, RFC930, M. Solomon, E. Wimmers

The TFTP Protocol, RFC783, K. R. Sollins

Time Protocol, RFC868, J. Postel, K. Harrenstien

Transmission Control Protocol, RFC793, J. Postel

Trivial File Transfer Protocol, RFC783, K. R. Sollins

User Datagram Protocol, RFC768, J. Postel

Related Information
In this book: "accept" in topic 1.2.9, "bind" in topic 1.2.20, "connect" in topic 1.2.49, "gethostbyaddr, gethostbyname, sethostent, endhostent" in

topic 1.2.98, "gethostid, sethostid" in topic 1.2.99, "gethostname, sethostname" in topic 1.2.100, "getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent" in topic 1.2.105, "getpeername" in topic 1.2.109, "getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent" in topic 1.2.112, "getservent, getservbyname, getservbyport, setservent, endservent" in topic 1.2.118, "getsockname" in topic 1.2.120, "getsockopt, setsockopt" in topic 1.2.121, "htonl, htons, ntohl, ntohs" in topic 1.2.131, "inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof" in topic 1.2.134, "listen" in topic 1.2.157, "rcmd, rresvport, ruserok" in topic 1.2.223, "recv, recvfrom, recvmsg" in topic 1.2.227, "resolver: res_mkquery, res_send, res_init, dn_comp, dn_expand, getshort, getlong, putshort, putlong" in topic 1.2.234, "rexec" in topic 1.2.235, "send, sendto, sendmsg" in topic 1.2.246, "shutdown" in topic 1.2.262, "socket" in topic 1.2.275, and "socketpair" in topic 1.2.276.

1.2.278 spools()

Purpose
Contains information to enable Transparent Computing Facility for UUCP communications facilities.

Description
The **/usr/adm/uucp/Spools** contains one or more lines, each using one of the following two formats:

```
remote master [local...]
remote master all
```

where the fields have the following meanings:

remote This is the name given to the remote uucp node.

master This is the name of the site within the TCF cluster that contains the master spool directory for this remote node.

local... These are optional sites in the cluster which can temporarily spool files queued to be delivered to the remote node.

all Specifies that all cluster sites can temporarily spool files queued to be delivered to the remote node.

Files

/usr/adm/uucp/Spools
<LOCAL>/spool/uucp

Related Information

The discussion about "Managing BNU" in the *Managing the Operating System*.

1.2.279 spropin

Purpose
Gets a more recent copy of a replicated file on the local file system.

Syntax

```
#include <sys/types.h>

spropin(gfs, inode_number, site_number)
gfs_t gfs;
ino_t inode_number;
siteno_t site_number;
```

Description
The **spropin** system call requests that a file's locally stored copy be brought up to date with respect to a copy at a remote site. The file is identified by its **gfs** and **inode_number** and the remote site is **site_number**. This is a call which is used to reconcile TCF replicated file systems. It should be used only by the superuser.

Return Value
Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **spropin** system call can fail with the following error codes:

EPERM The user is not superuser.

EINVAL Either **gfs** or **inode_number** has an improper value.

ENOSTORE Either the **gfs** is not mounted locally, it is not mounted remotely, or it has been corrupted.

EBADST The **site_number** is out of range or is the local site.

ELOCK Propagation is already in progress.

ENOSPC There is not enough room for the new version to be stored.

ESITEDN1 Lost contact with the remote site before propagation was complete.

EIO Read error on the remote site or write error on the local site.

Related Information

In this book: "chlwm" in topic 1.2.43 and "raccept" in topic 1.2.219.

The **primrec** and **recmstr** commands in *AIX Operating System Commands Reference*.

1.2.280 sputl, sgetl

Purpose
Accesses long numeric data in a machine-independent fashion.

Library
Object File Access Routine Library (**libld.a**)

Syntax

```
void sputl (value, buffer)  long sgetl (buffer)
long value;                 char *buffer;
char *buffer;
```

Description

The **sputl** subroutine stores the 4 bytes of the **value** parameter into memory starting at the location pointed to by the **buffer** parameter. The order of the bytes is the same across all machines.

The **sgetl** subroutine retrieves 4 bytes from memory starting at the location pointed to by the **buffer** parameter. It then returns the bytes as a **long** value with the byte ordering of the host machine.

Using **sputl** and **sgetl** subroutines together provides a machine-independent way of storing long numeric data in an ASCII file. For example, the numeric data stored in the portable archive file format is accessed with the **sputl** and **sgetl** subroutines.

Related Information
In this book: "frexp, ldexp, modf" in topic 1.2.85 and "ar" in topic 2.3.4.

1.2.281 ssignal, gsignal

Purpose
Implements a software signal facility.

Library
Standard C Library (**libc.a**)

Syntax

#include <signal.h>

```
void (*ssignal (sig, action)void)gsignal (sig)
int sig;                      int sig;
void (*action) ( );
```

Description
The **ssignal** and **gsignal** subroutines implement a software facility similar to that of the **signal** and **kill** system calls. However, there is no connection between the two facilities. User programs can use **ssignal** and **gsignal** to handle exceptional processing within an application. **signal** and related system calls handle system-defined exceptions.

The software signals available are associated with integers in the range 1 through 16. Other values are reserved for use by the C library and should not be used.

The **ssignal** subroutine associates the procedure specified by the **action** parameter with the software signal specified by the **sig** parameter. The **gsignal** subroutine "raises" the signal **sig**, causing the procedure specified by the **action** parameter to be taken.

The **action** parameter is either a pointer to a user-defined subroutine, or one of the constants **SIG_DFL** (default action) and **SIG_IGN** (ignore signal). The **ssignal** subroutine returns the procedure that was previously established for that signal. If no procedure was established before, or if the signal number is illegal, then **ssignal** returns the value **SIG_DFL**.

The **gsignal** subroutine "raises" the signal specified by the **sig** parameter by doing the following:

If the procedure for **sig** is **SIG_DFL**, then the **gsignal** subroutine returns a value of 0 and takes no other action.

If the procedure for **sig** is **SIG_IGN**, then the **gsignal** subroutine returns a value of 1 and takes no other action.

If the procedure for **sig** is a subroutine, then the **action** value is reset to **SIG_DFL** and the subroutine is called with **sig** passed as its parameter. The **gsignal** subroutine returns the value that is returned by the signal-handling subroutine.

If the procedure for **sig** is an illegal value or if no procedure was ever specified for that signal, then **gsignal** returns a value of 0 and takes no other action.

Related Information
In this book: "kill, kill3, killpg" in topic 1.2.138 and "sigaction,

sigvec, signal" in topic 1.2.263.

1.2.282 statx, fstatx, stat, fstat, fullstat, ffullstat, lstat

Purpose
Provides information about a file.

Syntax

```
#include <stat.h>

int statx(path, buf, len, cmd)
char *path;
struct stat *buf;
int len;
int cmd;

int fstatx(fildes, buf, len, cmd)
int fildes;
struct stat *buf;
int len;
int cmd;
```

Description
The **statx** and **fstatx** system calls obtain information about a file. The **path** parameter to **statx** is a path name identifying the file. The **fildes** parameter is a file descriptor obtained from a successful **open**, **fcntl**, **pipe**, **socket** or **socketpair** system call.

Information is returned in the **stat** structure pointed to by the **buf** parameter (see "stat.h" in topic 2.4.22). The **len** parameter indicates the amount of information to be returned.

The **cmd** parameter determines how to interpret the path name provided; specifically, whether to retrieve information about a symbolic link, hidden directory or mount point.

STX_LINK If **cmd** specifies **STX_LINK** and **path** is a path name which refers to a symbolic link, **statx** returns information about the symbolic link. Otherwise, **statx** returns information about the file to which the link refers.

If **cmd** specifies **STX_LINK** and **path** refers to a symbolic link, the **st_mode** and **st_type** fields of the returned **stat** structure indicates the file is a symbolic link.

STX_HIDDEN
If **cmd** specifies **STX_HIDDEN** and **path** is a path name which refers to a hidden directory, **statx** returns information about the hidden directory. Otherwise, **statx** returns information about the selected component of the hidden directory.

If **cmd** specifies **STX_HIDDEN** and **path** refers to a hidden directory, the **st_mode** and **st_type** fields of the returned **stat** structure indicate this is a hidden directory.

STX_MOUNT If **cmd** specifies **STX_MOUNT** and **path** names the root of a mounted file system, **statx** returns information about the mounted-over directory. Otherwise, **statx** returns information about the root of the mounted file system.

If **cmd** specifies **STX_MOUNT**, the **FS_MOUNT** bit in the st_flag

field of the returned **stat** structure is set if (and only if) this file is mounted over.

If **cmd** does not specify **STX_MOUNT**, the **FS_MOUNT** bit in the st_flag field of the returned **stat** structure is set if (and only if) this file is the root of a file system.

Subtopics
1.2.282.1 Compatibility Interfaces

1.2.282.1 Compatibility Interfaces

The following interfaces are provided for compatibility with programs written for AIX/RT or other versions of the UNIX operating system.

stat(path, stbuf)

is equivalent to:

statx(path, stbuf, STATSIZE, 0)

lstat(path, stbuf)

is equivalent to:

statx(path, stbuf, STATSIZE, STX_LINK)

fstat(fd, stbuf)

is equivalent to:

fstatx(fd, stbuf, STATSIZE, 0)

#include <sys/fullstat.h>

fullstat(path, cmd, buf)

is equivalent to:

statx(path, buf, FULLSTATSIZE, cmd)

#include <sys/fullstat.h>

ffullstat(fd, cmd, buf)

is equivalent to:

fstatx(fd, buf, FULLSTATSIZE, cmd)

Return Value
Upon successful completion, both the **statx** and **fstatx** system calls return a value of 0. If the **statx** or **fstatx** system calls fail, a value of -1 is returned, and **errno** is set to indicate the error.

Error Conditions
The **statx** and **fstatx** system calls fail if one or more of the following are true:

ENOTDIR A component of the path prefix is not a directory.

ENOENT A component of the path prefix does not exist, or the process has the **System V lookup** attribute and a component is exactly 14 characters long (see "ulimit" in topic 1.2.313).

EACCES Search permission is denied on a component of the path prefix.

ENOENT The path name is null.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

EFAULT The **path** parameter points to a location outside of the process's allocated address space.

ELOOP A loop of symbolic links was detected.

ENAMETOOLONG
A component of the **path** parameter exceeded **NAME_MAX** characters or the entire **path** parameter exceeded **PATH_MAX** characters.

ENOENT A hidden directory was named, but no component inside it matched the process's current site path list.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

EIO An I/O error occurred during the operation.

ENOENT The file named by **path** does not exist.

EBADF The **fildes** parameter is not a valid file descriptor.

If the Transparent Computing Facility is installed on your system, the **statx** and **fstatx** system calls can also fail if one or more of the following are true:

ESITEDN1 The **path** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE The **path** is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of **path** is replicated but is not stored on any site which is currently up.

ESITEDN1 The site or sites on which the file is stored are now down or the file descriptor is open for writing and the site where the file is stored has gone down since the file was opened.

ENFILE The system inode table on another cluster site is out of space.

EINTR A signal was caught during the **statx** system call.

Related Information
In this book: "chmod, fchmod" in topic 1.2.44, "chown, fchown" in topic 1.2.45, "link" in topic 1.2.156, "mknod, mknodx, mkfifo" in topic 1.2.169, "pipe" in topic 1.2.204, "read, readv, readx" in topic 1.2.224, "times" in topic 1.2.304, "unlink, rmslink, remove" in topic 1.2.318, "ustat" in topic 1.2.320, "utime" in topic 1.2.321, "write, writex" in topic 1.2.330, "master" in topic 2.3.32, and "stat.h" in topic 2.4.22.

Also see "The Base AIX File Systems" discussion in Chapter 1 of *Managing the AIX Operating System*.

1.2.283 stdio

Purpose
Performs standard buffered input and output operations.

Library
Standard I/O Library (**libc.a**)

Syntax

#include <stdio.h>

FILE *stdin, *stdout, *stderr;

Description
These macros and subroutines provide an efficient user-level I/O buffering scheme.

The in-line macros **getc** and **putc** handle characters quickly. The following macros and subroutines all use the **getc** and **putc** macros:

	getchar macro	**fread** subroutine
	getwchar macro	**fscanf** subroutine
	putchar macro	**fwrite** subroutine
	putwchar macro	**gets** subroutine
	fgetc subroutine	**getw** subroutine
	fgets subroutine	**getwc** subroutine
	fgetwc subroutine	**printf** subroutine
	fprintf subroutine	**puts** subroutine
	fputc subroutine	**putw** subroutine
	fputwc subroutine	**putwc** subroutine
	fputs subroutine	**scanf** subroutine
		wsprintf subroutine
		wsscanf subroutine

A file with associated buffering is called a ***stream*** and is declared to be a pointer to the defined type **FILE**. The **fopen** subroutine constructs descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the **stdio.h** header file and associated with the standard open streams:

stdin	Standard input stream
stdout	Standard output stream
stderr	Standard error output stream.

The constant NULL (0) designates a special pointer value that does not point to any data structure.

Most integer subroutines that deal with streams return the constant EOF (-1) upon end-of-file or an error. See each individual subroutine for detailed information about the return value.
Programs that use this input/output package must include the header file of pertinent macro definitions, as follows:

```
#include <stdio.h>
```

The subroutines and constants in the input/output package are declared in the header file and do not need any further declaration. The constants and the following routines are implemented as macros. Redeclaration of these names is not allowed.

	getc	**feof**
	getchar	**ferror**
	putc	**clearerr**
	putchar	**fileno**

Warning: Invalid stream pointers usually cause errors, possibly including program termination. Individual subroutine descriptions describe the possible error conditions.

Related Information
In this book: "close, closex" in topic 1.2.48, "ctermid" in topic 1.2.53, "cuserid" in topic 1.2.57, "fclose, fflush" in topic 1.2.77, "feof, ferror, clearerr, fileno" in topic 1.2.79, "fopen, freopen, fdopen" in topic 1.2.82, "fread, fwrite" in topic 1.2.84, "fseek, rewind, ftell" in topic 1.2.86, "getc, fgetc, getchar, getw, getwc, fgetwc, getwchar" in topic 1.2.91, "gets, fgets, getws, fgetws" in topic 1.2.117, "lseek" in topic 1.2.161, "open, openx, creat" in topic 1.2.199, "pipe" in topic 1.2.204, " popen, pclose, rpopen" in topic 1.2.207, "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf" in topic 1.2.208, "putc, putchar, fputc, putw, putwc, putwchar, fputwc" in topic 1.2.213, "puts, fputs, putws, fputws" in topic 1.2.216, "read, readv, readx" in topic 1.2.224, "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf, wsscanf" in topic 1.2.241, "setbuf, setvbuf" in topic 1.2.247, "system" in topic 1.2.298, "tmpfile" in topic 1.2.305, "tmpnam, tempnam" in topic 1.2.306, "ungetc, ungetwc" in topic 1.2.317, and "write, writex" in topic 1.2.330.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.284 stdipc: ftok

Purpose
Generates a standard interprocess communication key.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok (path, id)
char *path;
char id;
```

Description
The **ftok** subroutine returns a key, based on the **path** and **id** parameters, to be used to obtain interprocess communication identifiers. The **path** parameter must be the path name of an existing file that is accessible to the process. The **id** parameter must be a character that uniquely identifies a project. **ftok** returns the same key for linked files if called with the same **id** parameter. Different keys are returned for the same file if different **id** parameters are used.

All interprocess communication facilities require you to supply a key to the **msgget**, **semget**, and **shmget** system calls in order to obtain interprocess communication identifiers. The **ftok** subroutine provides one method of creating keys, but many others are possible. Another way to do this, for example, is to use the project ID as the most significant byte of the key, and to use the remaining portion as a sequence number.

Warning: It is important for each installation to define standards for forming keys. If some standard is not adhered to, unrelated processes can interfere with each other's operation.

If the **path** parameter of the **ftok** subroutine names a file that has been removed while keys still refer it, then the **ftok** subroutine returns an error. If that file is then recreated, the **ftok** subroutine will probably return a different key than the original one.

Return Value
Upon successful completion, the **ftok** subroutine returns a key that can be passed to the **msgget**, **semget**, or **shmget** system call. The **ftok** subroutine returns **(key_t) -1** if one or more of the following are true:

The file named by the **path** parameter does not exist.
The file named by the **path** parameter is not accessible to the process.
The **id** parameter is 0 (**'\0'**).

Related Information
In this book: "msgget" in topic 1.2.174, "semget" in topic 1.2.244, and "shmget" in topic 1.2.261.

1.2.285 stime

Purpose
Sets the time.

Syntax

```
int stime (tp)
time_t *tp;
```

Description
The **stime** system call sets the system's time and date. The **tp** parameter points to the time as measured in seconds from 00:00:00 GMT January 1, 1970.

Return Value
Upon successful completion, a value of 0 is returned. If the **stime** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **stime** system call fails if the following is true:

EFAULT Invalid pointer to **tp**.

EPERM The effective user ID of the calling process is not superuser.

Related Information
In this book: "time" in topic 1.2.303.

1.2.286 strcoll, strncoll, strxfrm, mbscoll, mbsncoll, wcscoll, wcsncoll

Purpose
Perform string and array comparisons using collating information.

Syntax

```
#include <string.h>


int strcoll (s1, s2)
const char *s1, *s2;

int strncoll (s1, s2, n)
const char *s1, *s2;
size_t n;

size_t strxfrm (s1, s2, n);
char *s1;
const char *s2;
size_t n;

int mbscoll (s1, s2)
const char *s1, *s2;

int mbsncoll (s1, s2)
const char *s1, *s2;

int wcscoll (s1, s2)
const wchar_t *s1, *s2;

int wcsncoll (s1, s2, n)
const wchar_t *s1, *s2;
size_t n;
```

Description
All of these subroutines except **strxfrm** perform file code and process code string and array comparisons using collating information. The **strxfrm** subroutine transforms strings. The **s1** and **s2** variables are pointers to file code strings.

The **strcoll** subroutine compares the string pointed to by **s1** to the string pointed to by **s2**, when both are interpreted according to the **LC_COLLATE** category of the current locale.

The **strncoll** subroutine compares the string pointed to by **s1** to the string pointed to by **s2** up to **n** characters, or when a terminating NULL is reached, when both strings are interpreted according to the **LC_COLLATE** category of the current locale.

The **strcoll** and the **strncoll** subroutines are identical to the **wcscoll** and the **wcsncoll** subroutines except that the latter compare process code strings instead of file code strings.

The **strxfrm** subroutine transforms the string pointed to by **s2** and places the resulting string into the array pointed to by **s1**. If the **strcmp** subroutine is called to compare the two transformed strings, **strxfrm** returns a value greater than, equal to or less than zero. This value corresponds to the result produced by calling the **strcoll** subroutine to compare the same two original strings. No more than **n** characters are placed into the resulting array pointed to by **s1**, including the terminating NULL character. If **n** is zero, **s1** is permitted to be a NULL pointer. If copying takes place between objects that overlap, the behavior is undefined.

The **mbscoll** subroutine compares the multibyte string pointed to by **s1** to the multibyte string pointed to by **s2**, when both are interpreted according to the **LC_COLLATE** category of the current locale.

The **mbsncoll** subroutine compares the multibyte string pointed to by **s1** to the multibyte string pointed to by **s2** up to **n** characters, or if a terminating NULL is reached, when both are interpreted according to the **LC_COLLATE** category of the current locale.

The **mbscoll** and **mbsncoll** subroutines have no shift states. Both input strings must begin in the initial shift state.

The **mbscoll** and **mbsncoll** subroutines are identical to the **wcscoll** and **wcsncoll** subroutines except that the latter compare process code strings instead of file code strings.

The **wcscoll** subroutine compares the array pointed to by **s1** to the array pointed to by **s2**, both interpreted as wide character codes converted from multibyte characters and in accordance with the **LC_COLLATE** category of the current locale.

The **wcsncoll** subroutine compares the array pointed to by **s1** to the array pointed to by **s2** up to **n** characters. Both strings are interpreted as wide character codes, converted from multibyte characters and in accordance with the **LC_COLLATE** category of the current locale.

Return Value

The **strcoll**, **strncoll**, **mbscoll**, and **mbsncoll** subroutines return an integer greater than, equal to, or less than zero depending on whether the string pointed to by **s1** is greater than, equal to or less than the string pointed to by **s2** when both are interpreted according to the **LC_COLLATE** category of the current locale.

The **strxfrm** subroutine returns the length of the transformed string (not including the terminating NULL character). If the value returned is **n** or more, the contents of the array pointed to by **s1** are indeterminate.

The **wcscoll** and **wcsncoll** subroutines return an integer greater than, equal to, or less than zero depending on whether the array pointed to by **s1** is greater than, equal to or less than the wide character array pointed to by **s2**, when both are interpreted according to the **LC_COLLATE** category of the current locale.

Error Conditions

The **strcoll**, **strncoll**, and **strxform** subroutines will fail if the following is true:

EINVAL The **s1** and **s2** arguments contain characters outside the domain of the collating sequence.

Related Information

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.287 strftime

Purpose
Places characters into an array.

Syntax

```
#include <time.h>

size_t strftime (s, maxsize, format, timeptr)
char *s;
size_t maxsize;
const char *format;
const struct tm *timeptr;
```

Description

The **strftime** subroutine places characters into the array pointed to by **s** as controlled by the string pointed to by **format**. The format will be a multibyte character sequence, beginning and ending in its initial shift state. The format string consists of zero or more conversion specifiers and ordinary multibyte characters. A conversion specifier consists of a percent sign (**%**) followed by a character that determines the behavior of the conversion specifier. All ordinary multibyte characters (including the terminating NULL character) are copied unchanged into the array.

If copying takes place between objects that overlap, the behavior is undefined. No more than *maxsize* characters are placed into the array. Each conversion specifier is replaced by appropriate characters as described by the **LC_TIME** category of the current locale and by the values contained in the structure pointed to by *timeptr*.

strftime parameters and their action

Flag	Action
%a	is replaced by the locale's abbreviated weekday name.
%A	is replaced by the locale's full weekday name.
%b	is replaced by the locale's abbreviated month name.
%B	is replaced by the locale's full month name.
%c	is replaced by the locale's appropriate date and time conversion.
%d	is replaced by the day of month as a decimal number (01-31.)
%D	is replaced by the date (%m/%d/%y).
%h	is replaced by the locale's abbreviated month name.

%H	is replaced by the hour (24-hour clock) as a decimal number (00-23).
%I	is replaced by the hour (12-hour clock) as a decimal number (01-12).
%j	is replaced by the day of the year as a decimal number (001-366).
%m	is replaced by the month as a decimal number (01-12).
%M	is replaced by the minute as a decimal number (00-59).
%n	is replaced by a newline character.
%p	is replaced by the locale's equivalent of the AM/PM designations associated with a 12-hour clock.
%r	is replaced by the time in a.m./p.m. notation according to British/US conventions (%I:%M:%S\[AM|PM]).
%S	is replaced by the second as a decimal number (00-61).
%t	is replaced by a tab character.
%T	is replaced by the time (%H:%M:%S:).
%U	is replaced by the week number of the year (Sunday as the first day of week 1) as a decimal number (00-53).
%w	is replaced by the weekday as decimal number [0(Sunday)-6]
%W	is replaced by the week number of the year (Monday as the first day of week 1) as a decimal number (00-53).
%x	is replaced by the locale's appropriate date representation.
%X	is replaced by the locale's appropriate time representation.
%y	is replaced by the year without century as a decimal number (00-99).
%Y	is replaced by the year with century as a decimal number.
%z	is replaced by the time zone name or abbreviation, or by no characters if no time zone is determinable.

%%	is replaced by %.

If a conversion specifier is not one of the above, a % is copied.

strftime parameters and their action.

Flag	**setlocale** token name	**C** locale default
%a	MBSDAY	Sun:Mon:Tue:Wed:Thu:Fri:Sat
%A	MBLDAY	Sunday:Monday:Tuesday:Wednesday
%b	MBSMONTH	Jan:Feb:Mar:Apr:May:Jun:Jul:Aug
%B	MBLMONTH	January:February:March:April:Ma
%c	MBLDATIM	%a %b %d %H:%M:%S %Z %Y
%p	MBAM_STR	a.m.
	MBPM_STR	p.m.
%x	MBLDATE	%b %d %Y
%X	MBTIME	%H:%M:%S

Return Value

If the total number of resulting characters including the terminating NULL character is not more than that of *maxsize*, the **strftime** function returns the number of characters placed into the array pointed to by **s**, not including the terminating null character. Otherwise, zero is returned and the contents of the array are indeterminate.

1.2.288 string

Purpose
Performs operations on strings.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <string.h>
```

```
char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
size_t n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
size_t n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
size_t n;
```

```
size_t strlen (s)
char *s;

char *strchr (s, c)
char *s;
int c;

char *strrchr (s, c)
char *s;
int c;

char *strpbrk (s1, s2)
char *s1, *s2;

size_t strspn (s1, s2)
char *s1, *s2;

size_t strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;
```

Description

Note: For subroutines which perform operations on wide and multibyte characters, refer to "wcstring" in topic 1.2.327 and "mbstring" in topic 1.2.164.

The **string** subroutines copy, compare, and append strings in memory, and they determine such things as location, size, and existence of strings in memory.

The parameters **s1**, **s2** and **s** point to strings. A string is an array of characters terminated by a NULL character. The subroutines **strcat**, **strncat**, **strcpy**, and **strncpy** all alter **s1**. They do not check for overflow of the array pointed to by **s1**. All string movement is performed character by character and starts at the left. Overlapping moves toward the left work as expected, but overlapping moves to the right may give unexpected results. All of these subroutines are declared in the **string.h** header file.

The **strcat** subroutine adds a copy of the string pointed to by the **s2** parameter to the end of the string pointed to by the **s1** parameter. The **strcat** subroutine returns a pointer to the NULL-terminated result.

The **strncat** subroutine copies at most **n** bytes of **s2** to the end of the string pointed to by the **s1** parameter. Copying stops before **n** bytes if a NULL character is encountered in the **s2** string. The **strncat** subroutine returns a pointer to the NULL-terminated result.

The **strcmp** subroutine lexicographically compares the string pointed to by the **s1** parameter to the string pointed to by the **s2** parameter. The **strcmp** subroutine uses native character comparison, which may be signed or unsigned. The **strcmp** subroutine returns a value that is:

Less than 0	If **s1** is less than **s2**
Equal to 0	If **s1** is equal to **s2**
Greater than 0	If **s1** is greater than **s2**.

The **strncmp** subroutine makes the same comparison as **strcmp**, but it compares at most **n** pairs of characters.

The **strcpy** subroutine copies the string pointed to by the **s2** parameter to the character array pointed to by the **s1** parameter. Copying stops when the NULL character is copied. The **strcpy** subroutine returns the value of the **s1** parameter.

The **strncpy** subroutine copies **n** bytes from the string pointed to by the **s2** parameter to the character array pointed to by the **s1** parameter. If **s2** is less than **n** characters long, then **strncpy** pads **s1** with trailing NULL characters to fill **n** bytes. If **s2** is **n** or more characters long, then only the first **n** characters are copied and the result is not terminated with a NULL character. The **strncpy** subroutine returns the value of the **s1** parameter.

The **strlen** subroutine returns the number of characters in the string pointed to by the **s** parameter, not including the terminating NULL character.

The **strchr** subroutine returns a pointer to the first occurrence of the character specified by the **c** parameter in the string pointed to by the **s** parameter. A NULL pointer is returned if the character does not occur in the string. The NULL character that terminates a string is considered to be part of the string.

The **strrchr** subroutine returns a pointer to the last occurrence of the character specified by the **c** parameter in the string pointed to by the **s** parameter. A NULL pointer is returned if the character does not occur in the string. The NULL character that terminates a string is considered to be part of the string.

The **strpbrk** subroutine returns a pointer to the first occurrence in the string pointed to by the **s1** parameter of any character from the string pointed to by the **s2** parameter. A NULL pointer is returned if no character matches.

The **strspn** subroutine returns the length of the initial segment of the string pointed to by the **s1** parameter that consists entirely of characters from the string pointed to by the **s2** parameter.

The **strcspn** subroutine returns the length of the initial segment of the string pointed to by the **s1** parameter that consists entirely of characters **not** from the string pointed to by the **s2** parameter.

The **strtok** subroutine returns a pointer to an occurrence of a text token

in the string pointed to by the **s1** parameter. The **s2** parameter specifies a set of token delimiters. If the **s1** parameter is anything other than NULL, then the **strtok** subroutine reads the string pointed to by the **s1** parameter until it finds one of the delimiter characters specified by the **s2** parameter. It then stores a NULL character into the string, replacing the delimiter, and returns a pointer to the first character of the text token. The **strtok** subroutine keeps track of its position in the string so that subsequent calls with a NULL **s1** parameter step through the string. The delimiters specified by the **s2** parameter can be changed for subsequent calls to **strtok**. When no tokens remain in the string pointed to by the **s1** parameter, the **strtok** subroutine returns a NULL pointer.

Related Information

In this book: "memory: memccpy, memchr, memcmp, memcpy, memset, bcopy" in topic 1.2.166, "NCstring" in topic 1.2.184, "NLstring" in topic 1.2.193, "mbstring" in topic 1.2.164, "wcstring" in topic 1.2.327, and "swab" in topic 1.2.292.

1.2.289 strstr

Purpose
Locates a substring.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <string.h>

char *strstr (string1, string2);
char *string1, *string2;
```

Description
The **strstr** function returns a pointer to the beginning of the first occurrence of **string2** in **string1**. The **strstr** function does not consider the null character (\0) that ends **string2** in the matching process.

If **string2** does not appear in **string1**, **strstr** returns NULL.

Example
The following example locates the string **hay** in the string **"needle in a haystack"**.

```
#include <string.h>

char *string1 = "needle in a haystack";
char *string2 = "hay";
char *result;

result = strstr (string1,string2);
   /* Result = a pointer to "hay" */
```

1.2.290 strtod, atof

Purpose
Converts an ASCII string to a floating-point number.

Library
Standard C Library (**libc.a**)

Syntax

```
double strtod (nptr, ptr)   double atof (nptr)
char *nptr, **ptr;          char *nptr;
```

Description
The **strtod** and **atof** subroutines convert a character string, pointed to by the **nptr** parameter, to a double-precision floating-point number. The first unrecognized character ends the conversion.

These subroutines recognize a character string when the characters appear in the following order:

1. An optional string of white-space characters
2. An optional sign
3. A string of digits optionally containing a decimal point
4. An optional **e** or **E** followed by an optionally signed integer.

This is represented symbolically as:

```
  {<sp>} [+|-] {<digit>} [.{digit}] [ (e|E) [+|-] {digit}]
```

where:

<sp> = whitespace
<digit> = 0-9
[ ] = 0 or 1
{ } = 0 or more

If the string begins with an unrecognized character, **strtod** and **atof** return the value 0.

If the value of **ptr** is not **(char **)** NULL, then a pointer to the character that terminated the scan is stored in ***ptr**. If an integer cannot be formed, ***ptr** is set to **nptr**, and 0 is returned.

If the correct return value overflows, **strtod** and **atof** return INF on AIX PS/2 and 0 on AIX/370. On underflow, **strtod** and **atof** return 0.

The **atof (nptr)** subroutine call is equivalent to **strtod (nptr, (char **)** NULL**)**.

The **strtod** and **atof** subroutines perform conversions to a floating-point number. See "strtol, atol, atoi" in topic 1.2.291 for information on conversions to integers.

Error Conditions
The **strtod** subroutine fails if the following is true:

ERANGE	The value to be returned would have caused overflow or underflow.

The **atof** subroutine may fail if the following is true:

ERANGE The correct value of the result would cause overflow or underflow.

Related Information

In this book: "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf, wsscanf" in topic 1.2.241 and "strtol, atol, atoi" in topic 1.2.291.

1.2.291 strtol, atol, atoi

Purpose
Converts a string to an integer.

Library
Standard C Library (**libc.a**)

Syntax

```
long strtol (str, ptr, base)long atol (str)
char *str, **ptr;           char *str;
int base;
                            int atoi (str)
                            char *str;
```

Description
The **strtol** subroutine returns a long integer whose value is represented by the character string **str**. **strtol** scans the string up to the first character that is inconsistent with the **base**. Leading whitespace characters are ignored.

Warning: Overflow conditions are ignored.

If the value of **ptr** is not **(char **)** NULL, a pointer to the character that terminated the scan is stored in ***ptr**. If an integer cannot be formed, ***ptr** is set to **str**, and 0 is returned.

If the **base** parameter is positive and not greater than 36, it is used as the base for conversion. After an optional leading sign, leading zeros are ignored. **0x** or **0X** is ignored if **base** is 16.

If the **base** parameter is 0, the string determines the base. Thus, after an optional leading sign, a leading 0 indicates octal conversion, and a leading **0x** or **0X** indicates hexadecimal conversion. The default is to use decimal conversion.

Note: Truncation from **long** to **int** can take place upon assignment, or by an explicit cast.

The **atol (str)** subroutine call is equivalent to **strtol (str, (char **)** NULL, 10**)**.

The **atoi (str)** subroutine call is equivalent to **(int) strtol (str, (char **)** NULL, 10**)**.

The **atoi** and **atol** subroutines do not actually call **strtol**.

The **strtol**, **atol**, and **atoi** subroutines perform conversions to integers. See "strtod, atof" in topic 1.2.290 for information on conversions to floating-point numbers.

Error Conditions
The **strol**, **atol**, and **atoi** subroutines fail if one of the following is true:

EINVAL The value of **base** is not supported.

ERANGE The value to be returned would cause an overflow.

Related Information
In this book: "strtod, atof" in topic 1.2.290 and "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf, wsscanf" in topic 1.2.241.

1.2.292 swab

Purpose
Copies bytes.

Library
Standard C Library (**libc.a**)

Syntax

```
void swab (from, to, nbytes)
short *from, *to;
int nbytes;
```

Description
The **swab** subroutine copies **nbytes** bytes from the location pointed to by the **from** parameter to the array pointed to by the **to** parameter, exchanging adjacent even and odd bytes.

The **nbytes** parameter should be even and nonnegative. If the **nbytes** parameter is odd and positive, the **swab** uses **nbytes**-1 instead. If the **nbytes** parameter is negative, then **swab** does nothing.

Related Information
In this book: "memory: memccpy, memchr, memcmp, memcpy, memset, bcopy" in topic 1.2.166 and "string" in topic 1.2.288.

1.2.293 swapctl

Purpose
Controls swap devices

Syntax

```
#include <sys/swap.h>

int swapctl(sc_type, sc_device)
int sc_type;
char *sc_device;

int swapctl(sc_type, sc_nrecs, sc_tab)
int sc_type, sc_nrecs;
swpt_t *sc_tab;
```

Description
The first form of **swapctl** adds a swap device to or deletes a swap device from the swapping subsystem. When **sc_type** is **SC_ADD**, the device specified by **sc_device** is added to the swapping subsystem and is used along with any other swap device that is a part of the swapping subsystem to page or swap memory pages to and from memory. When **sc_type** is **SC_DEL**, the device specified by **sc_device** is deleted from the swapping subsystem. A device that is deleted is not available for swapping.

The second form of **swapctl** provides information about swap devices. When **sc_type** is **SC_LIST**, information about **sc_nrecs** swap devices is placed in memory at the location specified by **sc_tab** and the number of available swap devices is returned. When **sc_nrecs** is 0, information about the swap devices is not placed in memory at the location specified by **sc_tab**; however, the number of available swap devices is returned.

Note: This system call is for use only by root processes.

Return Value
If **swapctl** fails, the return value is -1 and **errno** will be set to indicate the error.

Error Conditions
The **swapctl** system call fails if one or more of the following are true:

EFAULT Bad address.

EINVAL Invalid argument.

ELOCALONLY Operation restricted to local site.

ENODEV No such device.

ENOTBLK Block device required.

ENXIO No such device address.

EPERM Operation not permitted.

1.2.294 symlink

Purpose
Creates a symbolic link to a file or directory.

Library
Standard C Library (**libc.a**)

Syntax

```
int symlink (path1, path2)
char *path1, *path2;
```

Description
The **symlink** system call creates a symbolic link to a file or directory by creating a file that contains the name of the destination file to which it is being linked. The symbolic link functions as a pointer to the destination file or directory. The **path1** parameter specifies the destination file. If **path1** is not a full path name (does not begin with /), it is evaluated in the context of **path2**, not the current working directory (see "chdir" in topic 1.2.40.) No checks are made to determine if **path1** is a valid path name.

The **path2** parameter specifies the name of the symbolic link. Both path names are null-terminated character strings. If Network File System is installed on your system, these paths can cross outside your cluster into another node. A symbolic link can cross file system boundaries.

Use the **readlink** command to read the contents of the symbolic link.

If the destination file pointed to by the symbolic link is renamed or removed, the symbolic link points to a non-existent file. If the symbolic link itself is deleted, the destination file is not affected.

If **path1** begins with the path prefix **<LOCAL>/**, a process's local string set by the **setlocal** call will be substituted for this prefix when this symbolic link is later used. This allows a symbolic link to point to different files on different machines.

A symbolic link made in a system-level, replicated file system (refer to the description of replicated file systems in *Managing the AIX Operating System*) cannot be removed with a normal **unlink** call. The **rmslink** call must be used instead.

Symbolic links cannot be created in hidden directories (see "Hidden Directories" in topic 1.1.5.1.5).

Note: Indiscriminate use of symbolic links can confuse naive programs and users. Care should be used to avoid creating symbolic link loops.

Return Value
Upon successful completion, 0 is returned. Otherwise, a -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **symlink** system call fails if one or more of the following are true:

ENOTDIR A component of **path2** prefix is not a directory.

ENOENT A component of **path2** does not exist.

EACCES The requested link requires writing in a directory with a mode that denies write permission.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

EEXIST The link named by **path2** already exists.

ENOENT The **path2** parameter points to a NULL path name.

EACCES Search permission is denied on a component of the path prefix of **path2**.

EROFS The requested link requires writing in a directory on a read-only file system.

EFAULT The **path1** or **path2** parameter points to a location outside of the process's allocated address space.

ELOOP Too many symbolic links were encountered in translating **path2**.

ENFILE The system inode table is out of space.

ENOSPC The new symbolic link cannot be created because there is no space or no inodes left on the file system designated to contain the link.

EACCES An attempt was made to create a symbolic link inside a hidden directory.

EDQUOT The directory in which the entry for the new link is being placed cannot be extended, because the user's quota of disk blocks on the file system containing the directory has been exhausted.

If the Transparent Computing Facility is installed on your system, **symlink** can also fail if one or more of the following are true:

ESITEDN1 **path2** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE **path** is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of **path2** is replicated but is not stored on any site which is currently up.

EROFS The requested link requires writing in a replicated file system in which the primary copy is unavailable.

EINTR A signal was caught during the system call.

If Network File System is installed on your system, **symlink** can also fail if the following is true:

ESTALE The process's root or current directory is located in an NFS virtual file system that has been unmounted.

ETIMEDOUT The connection timed out.

Related Information

In this book: "Hidden Directories" in topic 1.1.5.1.5, "chdir" in topic 1.2.40, "link" in topic 1.2.156, "readlink" in topic 1.2.225, and "unlink, rmslink, remove" in topic 1.2.318.

The symbolic link section in *Using the AIX Operating System*.

The **ln** command in *AIX Operating System Commands Reference*.

1.2.295 sync

Purpose
Updates the super block, inodes, and delayed blocks.

Syntax

void sync ( )

Description
The **sync** system call causes all information in memory that should be on disk to be written out. The writing, although scheduled, is not necessarily complete upon return from the **sync** system call. Types of information to be written include modified super blocks, modified inodes, and delayed block I/O.

The **sync** system call should be used by programs that examine a file system, such as the **df** and **fsck** commands described in *AIX Operating System Commands Reference*.

Related Information
In this book: "fsync, fcommit" in topic 1.2.87.

The **sync** command in *AIX Operating System Commands Reference*.

1.2.296 sysconf

Purpose
Retrieves the value of a system limit or option.

Syntax

```
#include <unistd.h>

long sysconf(name)
int name;
```

Description
The **sysconf** system call allows an application to determine the characteristics of the system. This function allows an application program to determine at run time the presence of an optional facility or the value of one of the system-wide limits.

The **name** parameter specifies the configuration attribute to be queried. Symbolic values for the **name** parameter are defined in the file **unistd.h**.

Attribute	Meaning
_SC_ARG_MAX	Maximum length in bytes of arguments for the **exec** functions, including environment data.
_SC_CHILD_MAX	Maximum number of simultaneous processes on this machine per real user ID.
_SC_CLK_TCK	The number of intervals per second, used to express the value in **type clock_t**.
_SC_NGROUPS_MAX	Maximum number of simultaneous supplementary group IDs per process.
_SC_OPEN_MAX	Maximum number of files that one process can have open at any given time.
_SC_JOB_CONTROL	Returns the value 0, indicating that the job control facilities is provided in this system.
_SC_SAVED_IDS	Returns the value 0, indicating that **exec** functions save the effective user ID and effective group ID when a program starts. These saved IDs are then used for permission checking when being signalled, and when using the **setuid** system call.
_SC_VERSION	Returns the currently supported version of the POSIX standard.
_SC_TCF	Returns the value 0 if the Transparent Computing Facility is present in this system.
_SC_NFS	Returns the value 0 if the Network File System is present in this system.
_SC_VECPROC_SIZE	Returns the section size of the vector processor, if it is

present in this system. This parameter applies to the AIX/370 only.

Return Value
If the **sysconf** system call is successful, the specified parameter is returned. If the indicated system parameter is not set or if the specified parameter is not limited by the system, the value -1 is returned and **errno** is left unchanged. If the **name** is an unrecognized value, -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **sysconf** system call fails if the following is true:

EINVAL The value of the **name** argument is invalid.

Related Information
In this book: "pathconf, fpathconf" in topic 1.2.201.

1.2.297 syslog, openlog, closelog, setlogmask

Purpose
Makes a system log entry.

Library
Standard C Library (**libc.a**)

Syntax

#include <syslog.h>

```
int syslog (priority, messagint,closelog.([])
int priority;
char *message;              int setlogmask(maskpri)
                            int maskpri;
int openlog (ident, logopt, facility)
char *ident;
int logopt, facility;
```

Description
The **syslog** subroutine writes messages onto the system log maintained by the **syslogd** daemon. The message string *message* is similar to the **printf fmt** string, with the difference that **%m** is replaced by the current error message obtained from **errno**. A trailing new line can be added to the message if needed. The **val** parameters are the same as the **val** parameters of the **printf** subroutine. Each log message has a time stamp prepended to it.

Messages are read by the **syslogd** and written to the system console or log file, or forwarded to the **syslogd** on the appropriate host.

Messages are tagged with codes indicating the type of **priority** for each. A **priority** is encoded as a **facility**, which describes the part of the system generating the message, and as a level, which indicates the severity of the message.

The **facility** that generated the message is one of the following:

LOG_KERN Messages generated by the kernel. These cannot be generated by any user processes.

LOG_USER Messages generated by user processes. This is the default facility when none is specified.

LOG_MAIL The mail system.

LOG_DAEMON System daemons.

LOG_AUTH The authorization system: **login** and **su**, for example.

LOG_LPR The line printer spooling system.

LOG_LOCAL0 Reserved for local use.
--through--
LOG_LOCAL7

The **level** of severity is one of the following:

LOG_EMERG A panic condition reported to all users.

LOG_ALERT A condition that should be corrected immediately; for example, a corrupted database.

LOG_CRIT Critical conditions; for example, hard device errors.

LOG_ERR Errors.

LOG_WARNING Warning messages.

LOG_NOTICE Not an error condition, but a condition requiring special handling.

LOG_INFO General information messages.

LOG_DEBUG Messages containing information useful to debug a program.

If **syslog** cannot pass the message to **syslogd**, it writes the message on **/dev/console**, provided the **LOG_CONS** option is set.

If special processing is required, the **openlog** subroutine can be used to initialize the log file. The **ident** parameter contains a string that is attached to the beginning of every message. The default *ident* is syslog. The **facility** parameter encodes a default facility from the previous list to be assigned to messages that do not have an explicit facility encoded.

The **logopt** parameter is a bit field that indicates logging options. The values of **logopt** include:

LOG_PID Log the process ID with each message. This option is useful for identifying daemons.

LOG_CONS Send messages to the console if unable to send them to **syslogd**. This option is useful in daemon processes that have no controlling terminal.

LOG_NDELAY Open the connection to **syslogd** immediately, instead of when the first message is logged. This option is useful for programs that need to manage the order in which file descriptors are allocated.

LOG_NOWAIT Log messages to the console without waiting for forked child processes. Use this option for processes that enable notification of child process termination through **SIGCHLD**; otherwise, **syslog** may block, waiting for a child process whose exit status has already been collected.

The **closelog** subroutine closes the log file.

The **setlogmask** subroutine uses the bit mask in **maskpri** to set the new log priority mask and returns the previous mask. Logging is enabled for the levels indicated by the bits in the mask that are set and disabled where the bits are not set. The default mask allows all priorities to be logged.

The **LOG_MASK** and **LOG_UPTO** macros in the **/usr/include/sys/syslog.h** file are used to create the priority mask. Calls to **syslog** with a priority mask

that does not allow logging of that particular level of message cause the subroutine to return without logging the message.

Examples

```
  syslog (LOG_ALERT, "who:internal error 23");

  openlog ("ftpd", LOG_PID, LOG_DAEMON);
  setlogmask (LOG_UPTO (LOG_ERR));
  syslog (LOG_INFO, "Connection from host %d", CallingHost);

  syslog (LOG_INFO|LOG_LOCAL2, "foobar error:%m");
```

Related Information

In this book: "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf" in topic 1.2.208.

The discussion of **syslogd** in *AIX Operating System Commands Reference*.

1.2.298 system

Purpose
Runs a shell command.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <stdio.h>

int system (string)
char *string;
```

Description
The **system** subroutine passes the **string** parameter to the **sh** command as input. Then **sh** interprets **string** as a command and runs it.

The **system** subroutine invokes the **runl** system call to create a child process to run **/bin/sh**, which interprets the shell command contained in the **string** parameter. The current process waits until the shell has completed, then returns the exit status of the shell.

Note: The **system** subroutine runs only **sh** shell commands (also called Bourne shell commands). The results are unpredictable if the **string** parameter is not a valid **sh** shell command.

Return Value
The **system** subroutine is implemented using **runl** and **waitpid**. If either fails, a value of -1 is returned and **errno** is set to indicate the error.

File
/bin/sh

Error Conditions
The **system** subroutine fails if one or more of the following are true:

EAGAIN The system-imposed limit on the total number of processes under execution, system-wide or by a single user ID (**CHILD_MAX**), would be exceeded.

EINTR The **system** subroutine was interrupted by a signal.

ENOMEM Insufficient storage space is available.

Related Information
In this book: "exit, _exit" in topic 1.2.73, " run: runl, runv, runle, runve, runlp, runvp" in topic 1.2.239, and "wait, waitpid" in topic 1.2.325.

The **sh** command in *AIX Operating System Commands Reference*.

1.2.299 tcgetattr, tcsetattr

Purpose
Get and set terminal attributes.

Library
Standard I/O Library (**libc.a**)

Syntax

```
#include <termios.h>

int tcgetattr(fildes, termios_p)
int fildes;
struct termios *termios_p;

int tcsetattr(fildes, optional_actions, termios_p)
int fildes, optional_actions;
struct termios *termios_p;
int length;
```

Description
The **tcgetattr** function gets the parameters associated with the object to which **fildes** refers and stores them in the **termios** structure referenced by **termios_p**. This function is allowed from a background process. However, the information may be subsequently changed by a foreground process.

The **tcsetattr** function sets the parameters associated with the terminal from the **termios** structure referenced by **termios_p** as follows:

If **optional_actions** is **TCSANOW**, the change occurs immediately.

If **optional_actions** is **TCSADRAIN** the change occurs after all output written to **fildes** has been transmitted. The function should be used when changing parameters that affect output.

If **optional_actions** is **TCSAFLUSH**, the change occurs after all output written to the object to which **fildes** refers has been transmitted, and all input that has been received but not read is discarded before the change is made.

The **termios** structure below specifies the attributes which you can get or set with the **tcgetattr** and **tcsetattr** system calls. The sets of allowable attributes for various flags are defined in the file **termios.h** (see "termio" in topic 2.5.28) or descriptions of these flags.

```
    struct termios {
  tcflag_t          c_iflag;              /* terminal input control modes */
  tcflag_t          c_oflag;              /* terminal output control modes */
  tcflag_t          c_cflag;              /* hardware control of the terminal
  tcflag_t          c_lflag;              /* SYSV local modes */
  tcflag_t          c_reserved[4];        /* for future expansion */
  tcflag_t          c_bflag;              /* 4bsd: newtty line discipline mode
  struct _Winsize   c_winsize;            /* window size */
  char              c_length;             /* vertical screen length */
  char              c_pgflag;             /* paging and bell-ringing flags */
  char              c_line;               /* line discipline */
  cc_t              c_cc[NCCS];           /* control chars */
    };
```

Only the following fields of **c_bflag** can be set directly:

LCRTBS 0000001 Backspace on erase rather than echoing erase.

LPRTERA 0000002 Printing terminal erase mode.

LTILDE 0000010 Convert ~ to ' on output (for Hazeltine terminals).

LMDMBUF 0000020 Stop/start output when carrier drops.

LLITOUT 0000040 Suppress output translations.

LFLUSHO 0000200 Output is being flushed.

LETXACK 0001000 Diable style buffer hacking (unimplemented).

LCRTKIL 0002000 BS-space-BS erase entire line on line kill.

LCTLECH 0010000 Echo input control characters as ^X, delete as ^?.

LPENDIN 0020000 Retype pending input at next read or input character.

An attempt to set the other values of the **c_bflag** is ignored. Other values can be set via their **SysV** equivalents. The following describes the relationships existing between the **c_bflag** and the rest of the **SysV termios** flags:

LCRTERA set when **ECHOE** is set in **c_lflag**.

LNOHANG set when **CLOCAL** is set in **c_cflag**.

LNOFLASH set when **NOFLASH** is set in **c_lflag**.

LTOSTOP set when **TOSTOP** is set in **c_lflag**.

LDECCTQ set when **IXANY** is clear in **c_iflag** and start **char** is Ctrl-Q.

LPASS8 set when **CS8** is set in **c_cflag** and **ISTRIP** is clear in **c_iflag**.

When changing terminal attributes, an application should always do a **tcgetattr**, save the **termios** structure values returned, and then do a **tcsetattr**, changing only the necessary fields. The application should use the values saved from the **tcgetattr** to reset the terminal state whenever it is done with the terminal. This needs to be done because terminal attributes apply to the underlying port, not to each individual open instance. All processes that use the terminal see the latest attribute changes.

The **tcgetattr** and **tcsetattr** system calls are the preferred interfaces to the **tty** structure since they get and set all attributes, whereas the **SysV** and **BSD** interfaces only get and set a subset of attributes.

Return Value
Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **errno** is set to indicated the error.

Error Conditions
If any of the following conditions occur, the **tcgetattr** function returns -1 and sets **errno** to the corresponding value:

EBADF The **fildes** argument is not a valid file descriptor.

EINVAL The device does not support the **tcgetattr** function.

ENOTTY The file associated with **fildes** is not a terminal.

If any of the following conditions occur, the **tcsetattr** function returns -1 and sets **errno** to the corresponding value:

EBADF The **fildes** argument is not a valid file descriptor.

EINVAL The device does not support the **tcsetattr** function, the **optional_actions** argument is not a proper value, or an attempt was made to change an attribute represented in the termios structure to an unsupported value.

ENOTTY The file associated with **fildes** is not a terminal.

Related Information
In this book: "cfgetospeed, cfsetospeed, cfgetispeed, cfsetispeed" in topic 1.2.39, "tcsendbreak, tcdrain, tcflush, tcflow" in topic 1.2.301, and "tcgetpgrp, tcsetpgrp" in topic 1.2.300.

1.2.300 tcgetpgrp, tcsetpgrp

Purpose
Get and set foreground process group ID.

Library
Standard I/O Library (**libc.a**)

Syntax

```
#include <termios.h>

pid_t tcgetpgrp (fildes)
int fildes;

int tcsetpgrp (fildes, pgrp_id)
int fildes;
pid_t pgrp_id;
```

Description
The **tcgetpgrp** function returns the value of the process group ID of the foreground process group associated with the terminal. The **tcgetpgrp** function is allowed from a background process. However, the information may be subsequently changed by a foreground process.

The **tcsetpgrp** function sets the foreground process group ID associated with terminal to **pgrp_id** if the process has a controlling terminal. The file associated with **fildes** must be the controlling terminal of the calling process and the controlling terminal must be associated with the session of the calling process. The value of **pgrp_id** must not match a process ID or a process group ID of a process in another session.

Return Value
Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **errno** is set to indicated the error.

Error Conditions
If any of the following conditions occur, the function returns -1 and sets **errno** to the corresponding value:

EBADF The **fildes** argument is not a valid file descriptor.

EINVAL The device does not support the function.

ENOTTY The calling process does not have a controlling terminal or the file is not the controlling terminal.

EPERM The value of **pgrp_id** matches the process ID or process group ID of a process in another session.

Related Information
In this book: "cfgetospeed, cfsetospeed, cfgetispeed, cfsetispeed" in topic 1.2.39, "tcgetattr, tcsetattr" in topic 1.2.299, and "tcsendbreak, tcdrain, tcflush, tcflow" in topic 1.2.301.

1.2.301 tcsendbreak, tcdrain, tcflush, tcflow

Purpose
Line control functions.

Library
Standard I/O Library (**libc.a**)

Syntax

```
#include <termios.h>

int tcsendbreak (fildes, duration)
int fildes;
int duration;

int tcdrain (fildes)
int fildes;

int tcflush (fildes, queue_selector)
int fildes;
int queue_selector;

int tcflow (fildes, action)
int fildes;
int actions;
```

Description
If the terminal is using asynchronous serial data transmission, the **tcsendbreak** function causes transmission of a continuous stream of zero-valued bits for .25 seconds. The **duration** parameter is ignored.

The **tcdrain** function waits until all output written to the object to which **fildes** refers has been transmitted.

The **tcflush** function discards data written to the object to which **fildes** refers but not transmitted, or data received but not read, depending on the value of **queue_selector** as follows:

If **queue_selector** is **TCIFLUSH**, it flushes data received but not read.

If **queue_selector** is **TCOFLUSH**, it flushed data written but not transmitted.

If **queue_selector** is **TCIOFLUSH**, it flushes both data received but not read, and data written but not transmitted.

The **tcflow** function suspends transmission or reception of data on the object to which **fildes** refers, depending on the value of action as follows:

If action is **TCOOFF**, it suspends output.

If action is **TCOON**, it restarts suspended output.

If action is **TCIOFF**, the system transmits a **STOP** character, which causes the terminal device to stop transmitting data to the system.

If action is **TCION**, it restarts suspended output.

Return Value
Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **errno** is set to indicated the error.

Error Conditions
If any of the following conditions occur, the **tcsendbreak** function returns -1 and sets errno to the corresponding value:

EBADF The **fildes** argument is not a valid descriptor.

EINVAL The device does not support the **tcsendbreak** function.

ENOTTY The file associated with **fildes** is not a terminal.

If any of the following conditions occur, the **tcdrain** function returns -1 and sets **errno** to the corresponding value:

EBADF The **fildes** argument is not a valid file descriptor.

EINTR A signal interrupted the **tcdrain** function.

EINVAL The device does not support the **tcdrain** function.

ENOTTY The file associated with **fildes** is not a terminal.

If any of the following conditions occur, the **tcflush** function returns -1 and sets **errno** to the corresponding value:

EBADF The **fildes** argument is not a valid file descriptor.

EINVAL The device does not support the **tcflush** function, or **queue_selector** argument is not a proper value.

ENOTTY The file associated with **fildes** is not a terminal.

If any of the following conditions occur, the **tcflow** function returns -1 and sets **errno** to the corresponding value:

EBADF The **fildes** argument is not a valid file descriptor.

EINVAL The device does not support the **tcflush** function, or action argument is not a proper value.

ENOTTY The file associated with **fildes** is not a terminal.

Related Information
In this book: "cfgetospeed, cfsetospeed, cfgetispeed, cfsetispeed" in topic 1.2.39, "tcgetattr, tcsetattr" in topic 1.2.299, and "tcgetpgrp, tcsetpgrp" in topic 1.2.300.

1.2.302 termdef

Purpose
Queries terminal characteristics.

Library
Standard C Library (**libc.a**)

Syntax

```
char *termdef (fildes, c)
int fildes;
char c;
```

Description
The **termdef** subroutine returns a pointer to a null-terminated static character string that identifies a characteristic of the terminal that is open on the file descriptor specified by the **fildes** parameter. The **c** parameter specifies the characteristic that is to be queried. **termdef** determines this information by performing the following actions:

1. It queries the terminal device, using the **Query HFT Device** command, which is discussed on page 2.5.11.5.6.

2. If the query fails, then **termdef** uses the value of an environment variable.

3. If the environment variable is not set, then **termdef** returns the default value specified in the following table.

The following list shows the valid request types and the corresponding environment variables that are used if the **Query HFT Device** command fails:

c	Environment Variable	Default Value	Description
t	**TERM**	**"ibm5151"**	The terminal type
l	**LINES**	**NULL**	The number of lines or rows, based on the current font
c	**COLUMNS**	**NULL**	The number of character columns, based on the current font.

Note: When **fildes** identifies an asynchronous terminal, the **Query HFT Device** command always fails and the environment variable is always checked. The **TERM** variable is automatically set each time you log in. **LINES** and **COLUMNS** need to be set only if:

> You are using an asynchronous terminal and want to override the **lines** and **cols** settings in the **terminfo** data base, or
> Your asynchronous terminal has an unusual number of lines or columns and you are running an application that uses **termdef**, but not **terminfo**.

This is true because the **terminfo** initialization subroutine, **setupterm**, calls **termdef** to determine the number of lines and columns on the display. If **termdef** cannot supply this information, then **setupterm** uses the values in the **terminfo** data base.

Related Information
"Terminfo Level Subroutines" in topic 1.2.56.2, "terminfo" in

topic 2.3.59, and "Query HFT Device Command" in topic 2.5.11.5.6.

The **display** and **termdef** commands in *AIX Operating System Commands Reference*.

1.2.303 time

Purpose
Gets the time.

Syntax

```
#include <sys/types.h>
#include <time.h>


                time_t time ((time_t *) 0)  time_t time (tloc)
                                            time_t *tloc;
```

Description
The **time** system call returns the current time in seconds since 00:00:00 GMT, January 1, 1970.

If the **tloc** parameter is nonzero, the time is also stored in the location to which the **tloc** parameter points.

Return Value
Upon successful completion, the current time is returned. If the **time** system call fails, a -1 is returned and **errno** is set to indicate the error.

Error Conditions

EFAULT The **tloc** parameter points to a location outside of the process's allocated address space.

Related Information
In this book: "stime" in topic 1.2.285.

1.2.304 times

Purpose
Gets process and child process times.

Syntax

```
#include <sys/types.h>
#include <sys/times.h>

time_t times (buffer)
struct tms *buffer;
```

Description
The **times** system call fills the structure pointed to by the **buffer** parameter with time-accounting information. All time values reported by the **times** system call are in units of **CLK_TCK** **CLK_TCK** is defined in **<time.h>**.

The **tms** structure is defined in **sys/times.h** and it contains the following members:

```
clock_t tms_utime;
clock_t tms_stime;
clock_t tms_cutime;
clock_t tms_cstime;
```

This information comes from the calling process and each of its terminated child processes for which it has executed a **wait** system call.

tms_utime The CPU time used while executing instructions in the user space of the calling process.
tms_stime The CPU time used by the system on behalf of the calling process.
tms_cutime The sum of the **tms_utime**s and the **tms_cutime**s of the child processes.
tms_cstime The sum of the **tms_stime**s and the **tms_cstime**s of the child processes.

Note: The system measures time by counting clock interrupts. The precision of the values reported by the **times** system call depends on the rate at which the clock interrupts occur.

If the Transparent Computing Facility is installed, the reported times of a process that has been subject to a remote **exec** or **migrate** system call only reflect the time used since the process began running on the current site. The accumulated times from the previous sites are included in the accumulated times for child processes.

Return Value
Upon successful completion, the **times** system call returns the elapsed real time, in **CLK_TCK** units, since an arbitrary reference time in the past (for example, system start-up time). This reference time does not change from one call of **times** to another. If the **times** system call fails, a -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **times** system call fails if the following is true:

EFAULT The **buffer** parameter points to a location outside of the process's

allocated address space.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, " fork, vfork" in topic 1.2.83, "monitor, monstartup, moncontrol" in topic 1.2.171, "profil" in topic 1.2.210, "sysconf" in topic 1.2.296, "time" in topic 1.2.303, and "wait, waitpid" in topic 1.2.325.

The **cc** and **prof** commands in *AIX Operating System Commands Reference*.

1.2.305 tmpfile

Purpose
Creates a temporary file.

Library
Standard I/O Library (**libc.a**)

Syntax

#include <stdio.h>

FILE *tmpfile ( )

Description
The **tmpfile** subroutine creates a temporary file and returns its FILE pointer. The file is opened for update.

The temporary file is automatically deleted when the process using it terminates.

If the file cannot be opened, **tmpfile** writes an error message to the standard error output and returns a NULL pointer.

Error Conditions
The **tmpfile** subroutine may fail if the following are true:

EACCES Search permission is denied on a component of the path prefix of the file to be created, or write permission is denied for the parent directory of the file to be created.

EINTR A signal was caught during the **tmpfile** subroutine.

ENOMEM Insufficient storage space is available.

ENOTDIR A component of the path prefix of the file to be created is not a directory.

EROFS The file to be created would reside on a read-only file system.

Related Information
In this book: "fopen, freopen, fdopen" in topic 1.2.82, "mktemp" in topic 1.2.170, "open, openx, creat" in topic 1.2.199, " stdio" in topic 1.2.283, "tmpnam, tempnam" in topic 1.2.306, and "unlink, rmslink, remove" in topic 1.2.318.

1.2.306 tmpnam, tempnam

Purpose
Constructs the name for a temporary file.

Library
Standard I/O Library (**libc.a**)

Syntax

#include <stdio.h>

char *tmpnam (s)	**char *tempnam (dir, pfx)**
char *s;	**char *dir, *pfx;**

Description
The **tmpnam** and **tempnam** subroutines generate file names for temporary files.

The **tmpnam** subroutine generates a file name using the path name defined as **P_tmpdir** in the **stdio.h** header file. If the **s** parameter is NULL, the **tmpnam** subroutine places its result into an internal static area and returns a pointer to that area. The next call to this subroutine destroys the contents of the area.

If the **s** parameter is not NULL, it is assumed to be the address of an array of at least the number of bytes specified by **L_tmpnam**. **L_tmpnam** is a constant defined in **stdio.h**. The **tmpnam** subroutine places its results into that array and returns the value of the **s** parameter.

The **tempnam** subroutine allows you to control the choice of a directory. The **dir** parameter points to the path name of the directory in which the file is to be created. If the **dir** parameter is NULL or points to a string which is not a path name for an appropriate directory, the path name defined as **P_tmpdir** in the **stdio.h** header file is used. If that path name is not accessible, **/tmp** is used. You can bypass the selection of a path name by providing an environment variable, **TMPDIR**, in the user's environment. The value of the **TMPDIR** variable is a path name for the desired temporary-file directory. If the **TMPDIR** variable is used, both the **dir** parameter and **L_tmpnam** are ignored.

The **pfx** parameter of the **tempname** subroutine allows you to specify an initial character sequence with which the file name begins. The **pfx** parameter can be NULL, or it can point to a string of up to five characters to be used as the first few characters of the temporary file name.

The **tempnam** subroutine uses the **malloc** subroutine to obtain space for the constructed file name. The return value is a pointer to this space. Therefore, the pointer value returned by **tempnam** can be used as a parameter to the **free** subroutine.

If the **tempnam** subroutine cannot return the expected result for any reason (for example, if the **malloc** subroutine fails, or if an appropriate directory cannot be found), then it returns a NULL pointer.

Warning: The **tmpnam** and **tempnam** subroutines generate a different file name each time they are called. If they are called more than 4,096 times

by a single process, they start recycling previously used names.

File names created using these subroutines reside in a directory intended for temporary use, and their names are unique. It is your responsibility to use the **open** and **unlink** system calls to create the file and remove it when no longer needed.

These subroutines use the process ID of the process that invoked them. Therefore, another process cannot create, open, or close files of the same name unless it does not use these subroutines and explicitly builds file names using the current process ID of the process that created the file.

Error Conditions
The **tempnam** and **tmpnam** subroutines fail if the following is true:

ENOMEM Insufficient storage space is available.

Related Information
In this book: "fopen, freopen, fdopen" in topic 1.2.82, "malloc, free, realloc, calloc, valloc, alloca, mallopt, mallinfo" in topic 1.2.162, "mktemp" in topic 1.2.170, "open, openx, creat" in topic 1.2.199, "tmpfile" in topic 1.2.305, "unlink, rmslink, remove" in topic 1.2.318, and "environment" in topic 2.4.6.

1.2.307 trace_on

Purpose
Checks whether trace channel is enabled.

Library
Run-time Services Library (**librts.a**)

Syntax

int trace_on (chanmask)
unsigned long chanmask;

Description
The **trace_on** subroutine queries the application trace device driver to determine whether a given trace channel is enabled. **trace_on** allows a program to avoid the unnecessary overhead of setting up the trace message when its trace channel is disabled. **trace_on** is a C run-time subroutine and should be used by application programs, but not by device drivers.

The **chanmask** parameter is a mask with the bit corresponding to the channel number set. It can be formed by the expression **(1 << 31 - channum)**. User programs can use only channel number 31, which means that the value of **channum** must be **1** for user programs.

Making repeated calls to the trace device driver involves significant overhead, so call **trace_on** only once: either at the start of processing or just before the first trace point in the program.

If the application trace device driver is not already open, **trace_on** opens it.

Upon successful completion, **trace_on** returns 1 if the channel is enabled, or 0 if the channel is disabled. If the **trace_on** subroutine fails, a message is written to the standard error output, and a value of -1 is returned.

File
/dev/appltrace

Related Information
In this book: "trcunix" in topic 1.2.308, and "trace" in topic 2.5.29.

The **trace** command in *AIX Operating System Commands Reference*.

The discussion of **trace** in *AIX Programming Tools and Interfaces*.

1.2.308 trcunix

Purpose
Records application trace log entries.

Library
Run-time Services Library (**librts.a**)

Syntax

```
int trcunix (buf, cnt)
char *buf;
unsigned int cnt;
```

Description
The **trcunix** subroutine invokes the application trace device driver to record a trace log entry. **trcunix** is a C run-time subroutine. Device drivers should use the **trsave** subroutine to log trace events.

The **buf** parameter points to a buffer containing a 2-byte **traceid** followed by up to 20 bytes of user-defined trace data. The high-order 5 bits of the **traceid** specify the channel number, and the low-order 11 bits specify the hook ID for the message. User programs may use only channel number 31. The **cnt** parameter specifies the number of bytes in the buffer, including the **traceid**.

If the application trace device driver is not open, then **trcunix** opens it before writing the trace log entry to it.

Return Value
Upon successful completion, a value of 0 is returned and a trace log entry is written to **/dev/appltrace**. If the **trcunix** subroutine fails, an error message is written to the standard error output, and a value of -1 is returned.

File
/dev/appltrace

Related Information
In this book: "trace_on" in topic 1.2.307, and "trace" in topic 2.5.29.

The **trace** command in *AIX Operating System Commands Reference*.

The discussion of **trace** in *AIX Programming Tools and Interfaces*.

1.2.309 tsearch, tdelete, twalk

Purpose
Manages binary search trees.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <search.h>

char *tsearch (key, rootp, compar)
char *key;
char **rootp;
int (*compar) ( );

char *tdelete (key, rootp, compar)
char *key;
char **rootp;
int (*compar) ( );

void twalk (root, action)
char *root;
void (*action) ( );
```

Description
The **tsearch** subroutine performs a binary tree search. The algorithm is generalized from Donald E. Knuth's *The Art of Computer Programming,* Volume 3, 6.2.2, Algorithm T. (*) It returns a pointer into a tree indicating where the data specified by the **key** parameter can be found. If the data specified by the **key** parameter is not found, the data is added to the tree in the correct place. If there is not enough space available to create a new node, a NULL pointer is returned. The **rootp** parameter points to a variable that points to the root of the tree. If the variable to which *rootp* points is NULL, the variable is set to point to the root of a new tree.

The **compar** parameter is a pointer to the comparison function, which is called with two parameters that point to the elements being compared. The comparison function must compare its parameters and return a value as follows:

> If the first parameter is less than the second parameter, **compar** must return a value less than 0.
> If the first parameter is equal to the second parameter, **compar** must return 0.
> If the first parameter is greater than the second parameter, **compar** must return a value greater than 0.

The comparison function need not compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

If the **rootp** parameter is NULL on entry, then a NULL pointer is returned.

The **tdelete** subroutine deletes the data specified by the **key** parameter. It is generalized from Knuth (6.2.2) Algorithm D. The **rootp** and **compar** parameters perform the same function as they do for the **tsearch** subroutine. The variable pointed to by the **rootp** parameter will be changed if the deleted node is the root of the binary tree. The **tdelete**

subroutine returns a pointer to the parent node of the deleted node. If the data is not found, a NULL pointer is returned. If the **rootp** parameter is NULL on entry, then a NULL pointer is returned.

The **twalk** subroutine steps through the binary search tree whose root is pointed to by the **root** parameter. (Any node in a tree can be used as the root to step through the tree below that node.) The **action** parameter is the name of a routine to be invoked at each node. The routine specified by the **action** parameter is called with three parameters. The first parameter is the address of the node currently being pointed to. The second parameter is a value from an enumeration data type

```
typedef enum {preorder, postorder, endorder, leaf}  VISIT;
```

(This data type is defined in the **search.h** header file). The actual value of the second parameter depends on whether this is the first, second, or third time that the node has been visited during a depth-first, left-to-right traversal of the tree, or whether the node is a ***leaf***. A leaf is a node that is not the parent of another node. The third parameter is the level of the node in the tree, with the root node being level zero.

The pointers to the key and the root of the tree should be of type pointer-to-element and cast to type pointer-to-character. Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element. The **twalk** is a recursive function. A significant amount of user stack space must be allocated to use **twalk** on large tree structures.

Related Information
In this book: "bsearch" in topic 1.2.23, "hsearch, hcreate, hdestroy" in topic 1.2.130, and "lsearch, lfind" in topic 1.2.160.

(*) Reading, Massachusetts: Addison-Wesley, 1981.

1.2.310 ttyname, isatty, fullttyname

Purpose
Gets the name of a terminal.

Library
Standard C Library (**libc.a**)

Syntax

```
char *ttyname (fildes)          char *fullttyname (fildes)
int fildes;                     int fildes;
int isatty (fildes)
int fildes;
```

Description
The **ttyname** subroutine gets the name of a terminal. It returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor specified by the **fildes** parameter. A NULL pointer is returned if the file descriptor does not describe a terminal device in directory **/dev**.

If the Transparent Computing Facility is installed, the path name returned by **ttyname** includes the name of the **<LOCAL>** file system of the cluster site where the terminal is attached, if that site is not where **ttyname** is executing. For example, if the program calling **ttyname** is on site **prod** and the terminal is on site **manu**, the path name returned is **/manu/dev/tty02**. A NULL pointer is returned if the file descriptor does not describe a terminal device in the directory **<LOCAL>/dev** on any active cluster site.

The **fullttyname** subroutine is like **ttyname**, but the path name returned always includes the name of the **<LOCAL>** file system of the cluster site where the terminal is attached.

The **isatty** subroutine determines if the device associated with the file descriptor specified by the **fildes** parameter is a terminal. If the specified file descriptor is associated with a terminal, the **isatty** subroutine returns a value of 1. If the file descriptor is not associated with a terminal, a value of 0 is returned.

The return value of **ttyname** points to static data whose contents are overwritten by each call.

File
/dev/*

Error Conditions
The **ttyname** and **isatty** subroutines fail if one or more of the following are true:

EBADF The **fildes** argument is not a valid file descriptor.

ENOTTY The **fildes** argument does not refer to a terminal device.

1.2.311 ttysite

Purpose
Finds the site of a terminal.

Library
Standard C Library (**libc.a**)

Syntax

```
int ttysite(fildes)
int fildes;
```

Description
The **ttysite** subroutine returns the site number of the terminal device associated with the file descriptor **fildes**.

Related Information
In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137.

1.2.312 ttyslot

Purpose
Finds the slot in the **utmp** file for the current user.

Library
Standard C Library (**libc.a**)

Syntax

int ttyslot ( )

Description
The **ttyslot** subroutine returns the index of the current user's entry in the **/etc/utmp** file. The **ttyslot** subroutine scans the **/etc/utmp** file for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1, or 2).

The **ttyslot** subroutine returns -1 if an error was encountered while searching for the terminal name, or if none of the first three file descriptors (0, 1, and 2) is associated with a terminal device.

Warning: If the Transparent Computing Facility is installed, the process calling **ttyslot** must have its **<LOCAL>** file system equal to the **<LOCAL>** file system of the cluster site where the terminal associated with standard input, standard output or standard error is physically attached. Otherwise, incorrect results may be produced.

Files

/etc/inittab
/etc/utmp

Related Information
In this book: "getut: getutent, getutid, getutline, pututline, setutent, endutent, utmpname" in topic 1.2.126 and " ttyname, isatty, fullttyname" in topic 1.2.310.

1.2.313 ulimit

Purpose
Sets and gets user limits.

Syntax

```
#include <ulimit.h>

off_t ulimit (cmd, newlimit)
int cmd;
off_t newlimit;
```

Description
The **ulimit** system call controls process limits. The **cmd** parameter values are:

GET_FSIZE
Returns the process's file size limit. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

SET_FSIZE
Sets the process's file size limit to the value of the **newlimit** parameter. The limit is in units of 512-byte blocks. Any process can decrease this limit, but only a process with an effective user ID of superuser can increase the limit.

GET_DATALIM
Returns the maximum possible break value (see "brk, sbrk" in topic 1.2.21).

SET_DATLIM
Sets the maximum possible break value (see "brk, sbrk" in topic 1.2.21). Returns the new maximum break value, which is **newlimit** rounded up to the nearest page boundary.

GET_STACKLIM
Returns the lowest valid stack address. (Note that stacks grow from high addresses to low addresses.)

SET_STACKLIM
Sets the lowest valid stack address. Returns the new minimum valid stack address, which is **newlimit** rounded down to the nearest page boundary.

GET_REALDIR
Returns the current value of the **real directory read** flag. If this flag is 0, a **read** system call (or **readx** system call with **ext** of 0) against a directory returns fixed-format entries compatible with the System V UNIX Operating System. Otherwise, a **read** system call (or **readx** system call with **ext** of 0) against a directory returns the underlying physical format.

SET_REALDIR
Sets the value of the **real directory read** flag. If the **newlimit** parameter is 0, this flag is cleared; otherwise it is set. The old value of the **real directory read** flag is returned.

GET_SYSVLOOKUP

Returns the current value of the **System V lookup** flag. If this flag is nonzero, any **pathname** parameter containing a component of exactly 14 characters, passed to a system call, causes the system call to fail with the ENOENT error.

SET_SYSVLOOKUP
Sets the new value of the **System V lookup** flag. If the **newlimit** parameter is 0, this flag is cleared; otherwise it is set. The old value of the **System V lookup** flag is returned.

Return Value
Upon successful completion, a nonnegative value is returned. If the **ulimit** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **ulimit** system call fails and the limit remains unchanged if:

EPERM A process without superuser authority attempts to increase the file size limit.

EINVAL The **cmd** parameter is a value other than **GET_FSIZE**, **SET_FSIZE**, **GET_DATALIM**, **SET_DATLIM**, **GET_STACKLIM**, **SET_STACKLIM**, **GET_REALDIR**, **SET_REALDIR**, **GET_SYSVLOOKUP**, or **SET_SYSVLOOKUP**.

Example
To increase the size of the stack segment by 4096 bytes, and set **rc** to the new lowest valid stack address:

```
rc = ulimit(SET_STACKLIM, ulimit(GET_STACKLIM, 0) - 4096);
```

Related Information
In this book: "brk, sbrk" in topic 1.2.21, "getrlimit, setrlimit, vlimit" in topic 1.2.115, "read, readv, readx" in topic 1.2.224, "write, writex" in topic 1.2.330, and "master" in topic 2.3.32.

1.2.314 umask

Purpose
Sets and gets the value of the file creation mask.

Syntax

```
int umask (cmask)
int cmask;
```

Description
The **umask** system call sets the process's file mode creation mask to the value of the **cmask** parameter. Only the low-order 9 bits of the **cmask** parameter and the file mode creation mask are used.

Return Value
Upon successful completion, the previous value of the file mode creation mask is returned.

Related Information
In this book: "chmod, fchmod" in topic 1.2.44, "mknod, mknodx, mkfifo" in topic 1.2.169, "open, openx, creat" in topic 1.2.199, and "stat.h" in topic 2.4.22.

The **sh**, **csh**, and **umask** commands in *AIX Operating System Commands Reference*.

1.2.315 umount, fumount

Purpose
Unmounts a file system.

Syntax

```
int umount (dev)          int fumount  (dev)
char *dev;                char *dev;
```

Description
The **umount** and **fumount** system calls unmount a previously mounted file system contained on the block device (also called a ***special file***) identified by the **dev** parameter. The **dev** parameter is a pointer to a path name.

The **umount** system call does not unmount a file system if it contains open files. The **fumount** system call forces the file system to be unmounted even if it contains open files. The open files are closed by the system before unmounting and the file system is left in a clean state, ready for re-mounting. However, a file system is never unmounted if another file system is mounted on it.

If the Transparent Computing Facility is installed and the file system is replicated, the local copy may be unmounted if there are other copies of the file system still mounted.

After the file system is unmounted, the directory upon which the file system was mounted reverts to its ordinary interpretation as a directory.

The **umount** and **fumount** system calls can be invoked only by a process whose effective user ID is superuser.

Return Value
Upon successful completion, a value of 0 is returned. If the **umount** or **fumount** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **umount** or **fumount** system call fails if one or more of the following are true:

EPERM The process's effective user ID is not superuser.

ENOENT **dev** does not exist.

ENOTBLK **dev** is not the name of a block special file.

EINVAL **dev** is not mounted.

EINVAL **dev** is not local.

EBUSY A file on the device specified by the **dev** parameter is currently in use.

EFAULT **dev** points to a location outside of the process's allocated address space.

ENXIO **dev** is not currently configured.

ENAMETOOLONG
A component of the **dev** parameter exceeded **NAME_MAX** characters or the entire **dev** parameter exceeded **PATH_MAX** characters.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

ELOOP A loop of symbolic links was detected.

If the Transparent Computing Facility is installed on your system, **umount** or **fumount** can also fail if one or more of the following are true:

ENOSTORE **dev** is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of **dev** is replicated but not stored on any site which is currently up.

EINTR A signal was caught during the system call.

Related Information
In this book: "mount" in topic 1.2.172.

The **mount** and **umount** commands in *AIX Operating System Commands Reference*.

1.2.316 uname, unamex

Purpose
Gets the name of the current AIX system.

Syntax

```
#include <sys/utsname.h>

int uname (name)                int unamex (name)
struct utsname *name;           struct xutsname *name;
```

Description
The **uname** system call stores information identifying the current system in the structure pointed to by the **name** parameter.

The **uname** system call uses the **utsname** structure, which is defined in the **sys/utsname.h** file, and it contains the following members:

```
char  sysname[_UTSNMLEN];
char  nodename[_UTSNMLEN];
char  release[_UTSNMLEN];
char  version[_UTSNMLEN];
char  machine[_UTSNMLEN];
```

The **uname** system call returns a null-terminated character string naming the current system in the character array **sysname**. The **nodename** array contains the name that the system is known by on a communications network. The **release** and **version** arrays further identify the system.

The **machine** array identifies the CPU hardware being used. This array contains a null-terminated string which currently has one of the following values:

ID	Description
i386	PS/2
B370	System/370
XA370	System/370 XA

The **unamex** system call uses the **xutsname** structure, which is defined in the **sys/utsname.h** file, and it contains the following members:

```
unsigned long  nid;
long           reserved[3];
```

If the Transparent Computing Facility (TCF) is installed, the **xutsname.nid** field contains the TCF site number of the local machine; otherwise, the value of **xutsname.nid** is unspecified.

Return Value
Upon successful completion, a nonnegative value is returned. If the **uname** or **unamex** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **uname** and **unamex** system calls fail if:

EFAULT The **name** parameter points to a location outside of the process's allocated address space.

Related Information

The **uname** command in *AIX Operating System Commands Reference*.

1.2.317 ungetc, ungetwc

Purpose
Pushes a character or a wide character back into input stream.

Library
Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>

int ungetc (c, stream)
int c;
FILE *stream;

wchar_t ungetwc (c, iop)
wchar_t c;
FILE *iop;
```

Description
The **ungetc** subroutine inserts the character specified by the **c** parameter into the buffer associated with the input stream specified by the **stream** parameter. This causes the next call to the **getc** subroutine to return **c**. **ungetc** returns **c**, and leaves the **stream** file unchanged.

If the **c** parameter is EOF, then the **ungetc** subroutine does not place anything in the buffer and a value of -1 is returned.

You can always push one character back onto a stream, provided that something has been read from the stream or **setbuf** has been called. The **fseek** subroutine erases all memory of inserted characters.

The **ungetwc** subroutine pushes the wide character specified by **c** back onto the input stream pointed to by **stream**. The pushed-back wide characters are returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by **stream**) to a file positioning subroutine (**fseek**, **fsetopts** or **rewind**) discards any pushed-back wide characters for the stream. The external storage corresponding to the stream is unchanged.

One wide character of push-back is guaranteed. If the **ungetwc** subroutine is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

If the value of **c** equals that of the macro WEOF, the operation fails and the input stream is unchanged.

A successful call to the **ungetwc** subroutine clears the end-of-file indicator for the stream. The value of the file position indicator for the stream, after reading or discarding all pushed-back wide characters, is the same as it was before the wide characaters were pushed back. For a text stream, the value of its file position indicator after a successful call to the **ungetwc** subroutine is unspecified until all pushed-back wide characters are read or discarded. For a binary stream, the value of its file position indicator is undefined; if the value was zero before a call, it is indeterminate after the call.

Return Value
The **ungetc** subroutine returns -1 if it cannot insert the character.

The **ungetwc** subroutine returns the wide character pushed back after conversion, or WEOF if operation fails.

Related Information
In this book: "fseek, rewind, ftell" in topic 1.2.86, "getc, fgetc, getchar, getw, getwc, fgetwc, getwchar" in topic 1.2.91, "setbuf, setvbuf" in topic 1.2.247, and " stdio" in topic 1.2.283.

1.2.318 unlink, rmslink, remove

Purpose
Removes a directory entry.

Syntax

```
int unlink (path)          int remove (path)
char *path;                char *path;
int rmslink  (path)
char *path;
```

Description
The **unlink** system call removes the directory entry specified by the **path** parameter. The **remove** subroutine is identical to **unlink**. It is provided for ANSI-C compatability.

If the named file is a symbolic link to another file or directory, **unlink** removes the symbolic link, not the file or directory to which it refers.

When all links to a file are removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file are closed.

If *path* exists, and the parent directory of *path* has the *sticky* attribute bit set, the calling process must have an effective user ID equal to:

the owner ID of *path*, or to

the owner ID of the parent directory of *path*.

If the Transparent Computing Facility is installed, the following is applicable.

The system call **unlink** fails on symbolic links in system type replicated file systems. To remove the links, the **rmslink** system call must be used. This system call functions the same as **unlink** otherwise.

When a file which has been replicated is unlinked, the change is reflected in all other copies of the file system at the earliest possible moment. Copies of the file system which were not mounted when the unlink occurred, or mounted on a site not in communication with the primary site for the file system, do not see the effect of the unlink until that copy is mounted in a cluster that includes a copy of the file system which has seen the change.

Return Value
Upon successful completion, a value of 0 is returned. If the **unlink** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **unlink** system call fails and the named file is not unlinked if one or more of the following are true:

ENOTDIR A component of the path prefix is not a directory.

ENOENT The named file does not exist.

EACCES Search permission is denied for a component of the path prefix.

EACCES Write permission is denied on the directory containing the link to be removed.

EPERM The named file is a directory and the effective user ID of the process is not superuser.

EPERM The file named by the *path* parameter is in a directory with the *sticky* attribute bit set, and the effective user ID of the calling process is not equal to the owner of the file or of the parent directory.

EBUSY The entry to be unlinked is the mount point for a mounted file system.

ETXTBSY The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed.

EROFS The entry to be unlinked is part of a read-only file system.

EFAULT The **path** parameter points to a location outside of the process's allocated address space.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

ENAMETOOLONG
A component of the **path** parameter exceeded **NAME_MAX** characters or the entire **path** parameter exceeded **PATH_MAX** characters.

EISDIR A hidden directory was named, components within hidden directories must be explicitly named.

ENOENT A symbolic link was named in the path prefix, but the file to which it refers does not exist. Since **unlink** and **rmslink** do not follow symbolic links on the last component of a path, this error cannot occur on the last component.

ELOOP A loop of symbolic links was detected. Since **unlink** and **rmslink** do not follow symbolic links on the last component of a path, this error cannot occur on the last component.

ENFILE The system inode table is full.

If the Transparent Computing Facility is installed on your system, **unlink** can also fail if one or more of the following are true:

ESITEDN1 **path** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE **path** is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of **path** is replicated but not stored on any site which is currently up.

EROFS The directory entry to be unlinked is part of a replicated file

system whose primary copy is currently unavailable.

EINTR A signal was caught during the system call.

Related Information

In this book: "close, closex" in topic 1.2.48, "link" in topic 1.2.156, and "open, openx, creat" in topic 1.2.199.

The **rm** command in *AIX Operating System Commands Reference*.

1.2.319 usrinfo

Purpose
Gets and sets user information about the owner of the calling process.

Syntax

```
#include <uinfo.h>

int usrinfo (cmd, buf, count)
int cmd;
char *buf;
int count;
```

Description
The **usrinfo** system call gets and sets information about the owner of the current process. The information is a sequence of null-terminated **name=value** strings. The last string in the sequence is terminated by two successive null characters. A child process inherits the user information of its parent.

The **buf** parameter is a pointer to a user buffer. This buffer is usually **UINFOSIZ** bytes long.

The **count** parameter is the number of bytes of user information to be copied from or to the user buffer.

If the **cmd** parameter is one of the following constants:

GETUINFO Copies up to **count** bytes of user information into the buffer pointed to by the **buf** parameter.

SETUINFO Sets the user information for the process to the first **count** bytes in the buffer pointed to by the **buf** parameter. The effective user ID of the calling process must be superuser to set the user information.

The user information should at minimum consist of three strings that are typically set by the **login** program. These three strings are:

```
NAME=username
UID=userid
TTY=ttyname
```

If the process has no terminal, **ttyname** should be null.

Return Value
Upon successful completion, a nonnegative integer giving the number of bytes transferred is returned. If the **usrinfo** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **usrinfo** system call fails if one or more of the following are true:

EPERM The **cmd** parameter is set to **SETUINFO** and the effective user ID of the process is not superuser.

EINVAL The **cmd** parameter is not set to **SETUINFO** or **GETUINFO**.

EINVAL The **cmd** parameter is set to **SETUINFO** and the count parameter is

larger than **UINFOSIZ**.

EFAULT The **buf** parameter points to a location outside of the process's allocated address space.

Related Information
In this book: "getuinfo" in topic 1.2.125.

The **login** command in *AIX Operating System Commands Reference*.

1.2.320 ustat

Purpose
Gets file system statistics.

Syntax

```
#include <sys/types.h>
#include <ustat.h>

int ustat (dev, buf)
dev_t dev;
struct ustat *buf;
```

Description
The **ustat** system call gets information about the mounted file system identified by device number **dev**. The **dev** parameter is the ID of the file system, and it corresponds to the **st_dev** member of the structure returned by the **statx**, **stat**, and **fullstat** system calls.

If the Transparent Computing Facility is installed, the following warning is applicable.

Warning: When **ustat** is used on a replicated file system, the space on the local copy is returned, if there is a local copy; otherwise, space information from the current synchronization site is returned.

See "dustat" in topic 1.2.66 for more information on obtaining data for replicated file systems. See "statx, fstatx, stat, fstat, fullstat, ffullstat, lstat" in topic 1.2.282 for more information about the device ID. The information is stored into a structure pointed to by the **buf** parameter.

The **ustat** structure pointed to by the **buf** parameter is defined in the **ustat.h** file, and it contains the following members:

```
daddr_t   f_tfree;        /* Total free blocks */
                          /* in units of 512 byte blocks */
ino_t     f_tinode;       /* Number of free inodes */
char      f_fname[6];     /* File system name */
char      f_fpack[6];     /* File system pack name */
```

Return Value
Upon successful completion, a value of 0 is returned. If the **ustat** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **ustat** system call fails if one or more of the following are true:

EINVAL **dev** is not the device number of a device containing a mounted file system.

EFAULT The **buf** parameter points to a location outside of the process's allocated address space.

If the Transparent Computing Facility is installed on your system, **ustat** can also fail if one or more of the following are true:

ESITEDN1 The request cannot be serviced because the site with the file

system is no longer available.

ESITEDN2 The operation was terminated because a site failed.

EINTR A signal was caught during the system call.

Related Information
In this book: "dustat" in topic 1.2.66, "statx, fstatx, stat, fstat, fullstat, ffullstat, lstat" in topic 1.2.282, and "fs" in topic 2.3.20.

1.2.321 utime

Purpose
Sets file access and modification times.

Library
Berkeley Compatability Library (**libbsd.a**)

Syntax

```
#include <unistd.h>

int utime (path, times)
char *path;
struct utimbuf *times;
```

Description
The **utime** system call sets the access and modification times of the file pointed to by the **path** parameter to the value of the **times** parameter. The inode changed time is set to the current time.

If the **times** parameter is NULL, the access and modification times of the file are set to the current time. If the file is a remote file, the current time at the local node-- not the remote node--is used. The effective user ID of the process must be the same as the owner of the file or must have write permission or superuser authority in order to use the **utime** system call in this manner.

If the **times** parameter is not NULL, it is a pointer to a **utimbuf** structure and the access and modification times are set to the values contained in the designated structure, regardless of whether or not those times correlate with the current time. Only the owner of the file or superuser can use the **utime** system call this way.

The **utimbuf** structure pointed to by the **times** parameter is defined in the **unistd.h** file, and it contains the following members.

```
time_t actime;    /* Date and time of last access */
time_t modtime;   /* Date and time of last modification */
```

The times in this structure are measured in seconds since 00:00:00 GMT, January 1, 1970.

Return Value
Upon successful completion, a value of 0 is returned. If the **utime** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **utime** system call fails if one or more of the following are true:

ENOENT The named file does not exist.

ENOENT A hidden directory was named, but no component inside it matched the process's current site path list.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

ENOTDIR A component of the path prefix is not a directory.

EACCES Search permission is denied by a component of the path prefix.

EPERM The effective user ID is not superuser or the owner of the file and the **times** parameter is not NULL.

EACCES The effective user ID is not superuser or the owner of the file, the **times** parameter is NULL, and write access is denied.

EROFS The file system containing the file is mounted read-only.

EFAULT The **times** or **path** parameter points to a location outside of the process's allocated address space.

ESTALE The process's root or current directory is located in an NFS virtual file system that has been unmounted.

ENAMETOOLONG
A component of the **path** parameter exceeded **NAME_MAX** characters or the entire **path** parameter exceeded **PATH_MAX** characters.

ELOOP A loop of symbolic links was detected.

ETXTBSY The **times** parameter is NULL and the file is a pure procedure (shared text) file that is being executed.

If the Transparent Computing Facility is installed on your system, **utime** can also fail if one or more of the following are true:

ESITEDN1 **path** cannot be accessed because a site went down.

ESITEDN2 The operation was terminated because a site failed.

ENOSTORE **path** is a name relative to the working directory, but no site which stores this directory is currently up.

ENOSTORE A component of **path** is replicated but not stored on any site which is currently up.

EROFS The named file resides on a replicated file system in which the primary copy is unavailable.

EINTR A signal was caught during the system call.

Related Information
In this book: "statx, fstatx, stat, fstat, fullstat, ffullstat, lstat" in topic 1.2.282.

1.2.322 utimes

Purpose
Sets file times.

Library
Berkeley Compatability Library (**libbsd.a**)

Syntax

```
#include <sys/time.h>

int utimes (file, tvp)
char *file;
struct timeval tvp[2];
```

Description
The **utimes** system call sets the accessed and modification times of a file to the values specified by the **tvp** parameter. The inode-changed time of the file is set to the current time.

The calling process must be the owner of the file or have an effective user ID of superuser.

Note: In the AIX Operating System, file times have a resolution of one second.

Return Value
When the call succeeds, a value of 0 is returned. If **utimes** fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **utimes** system call fails if one or more of the following is true:

EACCES Search permission is denied to a component of the path prefix.

EFAULT The **file** or **tvp** parameter points to a location outside of the process's allocated address space.

ENOENT The named file does not exist.

ENOTDIR A component of the path prefix is not a directory.

EPERM The calling process is not the owner of the file and does not have an effective user ID of superuser.

EROFS The file system that contains the file is mounted read-only.

ENAMETOOLONG
A component of the **path** parameter exceeded **NAME_MAX** characters or the entire **path** parameter exceeded **PATH_MAX** characters.

ENOENT A hidden directory was named, but no component inside it matched the process's current site path list.

ENOENT A symbolic link was named, but the file to which it refers does not exist.

ELOOP A loop of symbolic links was detected.

ETXTBSY The **path** parameter is pure procedure (shared text) file that is being executed.

Related Information

In this book: "statx, fstatx, stat, fstat, fullstat, ffullstat, lstat" in topic 1.2.282.

1.2.323 varargs

Purpose
Handles a variable-length parameter list.

Library
Standard C Library (**libc.a**)

Syntax

```
#include <varargs.h>

va_alist                        type va_arg (argp, type)
                                va_list argp;
va_dcl
                                void va_end (argp)
void va_start (argp)            va_list argp;
va_list argp;
```

Description
This set of macros allows you to write portable subroutines that accept a variable number of parameters. Subroutines that have variable-length parameter lists (such as **printf**), but that do not use **varargs**, are inherently nonportable because different systems use different parameter-passing conventions.

va_alist Is used as the parameter list in the function header.

va_dcl Is the declaration for **va_alist**. No semicolon should follow **va_dcl**.

va_list Defines the type of the variable used to traverse the list.

va_start Initializes **argp** to point to the beginning of the list.

argp Is a variable that the **varargs** macros use to keep track of the current location in the parameter list. Do not modify this variable.

va_arg Returns the next parameter in the list pointed to by **argp**. **type** is the data type that the parameter is expected to be. Different types can be mixed, but your subroutine must know what type of parameter is expected because it cannot be determined at runtime. The **printf** subroutine solves this problem by using its **format** parameter to determine the parameter types expected.

va_end Cleans up at the end.

Your subroutine can traverse, or scan, the parameter list more than once. Start each traversal with a call to **va_start** and end it with **var_end**.

Note: The calling routine is responsible for specifying the number of parameters because it is not always possible to determine this from the stack frame. For example, **execl** is passed a NULL pointer to signal the end of the list. **printf** determines the number of parameters from its **format** parameter.

Specifying **char**, **short**, or **float** as the second parameter to **va_arg**

is not portable because parameters seen by the called subroutine are not **char**, **short**, or **float**. The C complier converts **char** and **short** parameters to **int**, and it converts **float** parameters to **double** before passing them to a subroutine.

Example

The following code example is a possible alternate implementation of **execl** system call:

```
#include <varargs.h>

#define MAXARGS 100
/*
**  execl is called by
**  execl(file, arg1, arg2,..., (char *) 0);
*/
execl(va_alist)
   va_dcl
{
   va_list ap;
   char *file;
   char *args[MAXARGS];
   int argno = 0;

   va_start(ap);
   file = va_arg(ap, char *);
   while ((args[argno++] = va_arg(ap, char *)) != (char *) 0)
      ;   /* Empty loop body */
   va_end(ap);
   return (execv(file, args));
}
```

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf" in topic 1.2.208, and "vprintf, vfprintf, vsprintf, NLvprintf, NLvfprintf, NLvsprintf" in topic 1.2.324.

1.2.324 vprintf, vfprintf, vsprintf, NLvprintf, NLvfprintf, NLvsprintf

Purpose

Formats a **varargs** parameter list for output.

Library

Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>
#include <varargs.h>
```

```
int vprintf (format, argp)  int NLvprintf (format, argp)
char *format;               char *format;
va_list argp;               va_list argp;

int vfprintf (stream, formatintrNLvfprintf (stream, format, ar
FILE *stream;               FILE *stream;
char *format;               char *format;
va_list argp;               va_list argp;

int vsprintf (s, format, argNLvsprintf (s, format, argp)
char *s, *format;           char *s, *format;
va_list argp;               va_list argp;
```

Description

The **vprintf**, **vfprintf**, and **vsprintf** subroutines format and write **varargs** parameter lists. They are the same as the **printf**, **fprintf**, and **sprintf** subroutines, respectively, except that they are not called with a variable number of parameters. Instead, they are called with a parameter list pointer as defined by "varargs" in topic 1.2.323.

The **NLvprintf**, **NLvfprintf**, and **NLvsprintf** subroutines are provided for backward compatibility and behave exactly like the **vprintf**, **vfprintf**, and **vsprintf** subroutines respectively.

Example

The following example demonstrates how the **vfprintf** subroutine could be used to write an error routine.

```
#include <stdio.h>
#include <varargs.h>

/* error should be called with the syntax:       */
/* error(routine_name, format [, value,...]); */

/*VARARGS0*/

void error(va_alist)
va_dcl
/*
**  Note that the function name and format arguments
**  cannot be separately declared because of the
**  definition of varargs.
*/
{
   va_list args;
```

```
    char *fmt;

    va_start(args);
    /*
    ** Display the name of the function that called error
    */
    (void) fprintf(stderr, "ERROR in %s: ", va_arg(args, char *));
    /*
    ** Display the remainder of the message
    */
    fmt = va_arg(args, char *);
    (void) vfprintf(std err, fmt, args);
    va_end(args);
    (void) abort();
}
```

Error Conditions
The **vprintf** subroutine fails if one or more of the following are true:

EAGAIN The **O_NONBLOCK** flag is set for the file descriptor underlying **stream** and the process is delayed in the write operation.

EBADF The file descriptor underlying **stream** is not a valid file descriptor open for writing.

EFBIG An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.

EINTR The write operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfers for this file.

EIO The implementation supports job control, the process is a member of a background process group attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking **SIGTTOU** and the process group of the process is orphaned.

ENOSPC There was no free space remaining on the device containing the file.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device.

EPIPE An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

ENOMEM Insufficient storage space is available.

EINVAL There are insufficient arguments.

Related Information
In this book: "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf" in topic 1.2.208.

1.2.325 wait, waitpid

Purpose
Obtains status information pertaining to a child process.

Syntax

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

pid_t wait(NULL);

pid_t wait(stat_loc);
int *stat_loc;

pid_t waitpid(pid, stat_loc, options);
pid_t pid;
int stat_loc;
int options;
```

Description
The **wait** and **waitpid** system calls allow the calling process to obtain status information pertaining to one of its child processes. Various options permit the **waitpid** system call to be used to obtain status information for child processes that have terminated or stopped. If status information is available for more than one child process, the order in which this status is reported is not reliable.

The **wait** system call suspends execution of the calling process until status information for one of its terminated child processes is available, or until a signal is delivered (to the calling process) whose action is either to execute a signal catching function or to terminate the process. If status information is available prior to the call to the **wait** system call, the **wait** system call immediately returns with status information. This is not to say that the **wait** system call will be executed atomically; other processes may execute between the time the call to the **wait** system call is made and the time that the **wait** system call returns.

If the **waitpid** system call is given a **pid** argument with a value of -1 and an **options** argument with a value of 0, then the **waitpid** system call behaves identically to the **wait** system call. Otherwise, the behavior of the **waitpid** system call is similar to the behavior of the **wait** system call, but differs as indicated by the values of its **pid** and **options** arguments.

The **pid** argument specifies the set of child processes for which status may be reported by the **waitpid** system call. The **waitpid** system call does not return the status of a child process that is not in the set specified by the **pid** argument.

A **pid** argument with a value that is less than -1 specifies the set which includes all child processes with a process group ID that is equal to the absolute value of the **pid** argument. A **pid** argument with a value of -1 specifies the set which includes all child processes. This is the same set that is implicitly specified in a **wait** system call. A **pid** argument with a value of 0 specifies the set which includes all child processes that are in the same process group as the process which called the **waitpid** system call. A **pid** argument with a value that is greater than 0 specifies the set which includes only the child process with a process ID that is

equal to the value of the **pid** argument.

The **options** argument is constructed from the bitwise inclusive OR of 0 or more of the **WNOHANG** and **WUNTRACED** flags. These flags are defined in the header **<sys/wait.h>**.

If the **WNOHANG** flag is inclusive ORed into the **options** argument of a **waitpid** system call, the lack of immediately available status information (for one of the processes that is in the set specified by **pid**) will not cause the suspension of the execution of the calling process.

If the **WUNTRACED** flag is inclusive ORed into the **options** argument of a **waitpid** system call, the status of any child processes in a set of child processes specified by the **pid** argument (of the same call to the **waitpid** system call) that are stopped, and whose status has not yet been reported since they stopped will be included in the set of processes from which one member has its status reported to the calling process.

If the **wait** or **waitpid** system call returns to report the status of a child process, the call returns the value of the process ID of the child process whose status is being reported. In the case in which the **wait** or **waitpid** system call returns to report the status of a child process, if the value of the argument **stat_loc** is not NULL, **wait** and **waitpid** store information in ***stat_loc** (that is, the location pointed to by **stat_loc**). The **wait** and **waitpid** calls store a value of 0 at ***stat_loc** only if the status that the **wait** or **waitpid** system call reports is from a terminated child process that returned a value of 0 from its main or gave a status argument with a value of 0 to a call to **_exit** or **exit**. Any value that **wait** or **waitpid** stores in ***stat_loc** may be interpreted using the macros **WIFEXITED**, **WEXITSTATUS**, **WIFSIGNALED**, **WTERMSIG**, **WIFSTOPPED**, and **WSTOPSIG**. These macros are defined in <**sys/wait.h**> and evaluate to values of type **int**. Note that these macros do not accept a constant as an argument.

The macro **WIFEXITED(*stat_loc**) evaluates to a nonzero value if status was reported for a child process that terminated normally.

If the macro **WIFEXITED(*stat_loc**) evaluates to a nonzero value, the macro **WEXITSTATUS(*stat_loc**) evaluates to the value of the low-order 8 bits of the status argument that the child process (for which status was reported) returned from its **main** or gave as the status argument of a call to **_exit** or **exit**.

The macro **WIFSIGNALED(*stat_loc**) evaluates to a nonzero value if status was reported for a child process that terminated due to the receipt of a signal that was not caught.

If the macro **WIFSIGNALED(*stat_loc**) evaluates to a nonzero value, the macro **WTERMSIG(*stat_loc**) evaluates to the number of the signal that caused the termination of the child process for which status was reported.

The macro **WIFSTOPPED(*stat_loc**) evaluates to a nonzero value if status was reported for a child process because that child process was stopped.

If the macro **WIFSTOPPED(*stat_loc**) evaluates to a nonzero value, then the macro **WSTOPSIG(*stat_loc**) evaluates to the number of the signal that caused the child process (for which status was reported) to stop.

If the information stored in ***stat_loc** was stored there by a call to the **waitpid** system call that was passed an **options** argument that had **WUNTRACED** bitwise inclusive ORed into it, exactly one of the macros

WIFEXITED(*stat_loc), **WIFSIGNALED(*stat_loc**), and **WIFSTOPPED(*stat_loc**) evaluates to a nonzero value. If the information stored in ***stat_loc** was stored there by a call to the **waitpid** system call that was passed an **options** argument that did not have **WUNTRACED** bitwise inclusive ORed into it or by a call to the **wait** system call, exactly one of the macros **WIFEXITED(*stat_loc**) and **WIFSIGNALED(*stat_loc**) evaluates to a nonzero value.

The following information may also be used to interpret ***stat_loc**:

If the child process stopped in a trace mode, then bits 8-15 (startin with the least significant bit) of ***stat_loc** contain the number of the signal that caused the process to stop and the low-order 8 bits are set to one of the following:

- 0177 (0x7F), which indicates the process was stopped while being traced. If multi-process debugging mode was not set by the **ptrace** system call, the low-order bits are set to this value. (For more information on multi-process debugging, see "ptrace" in topic 1.2.212.)

- 0176 (0x7E), which means the process was stopped after a **fork** system call. Both the traced process and its newly forked child process are stopped. This value only occurs when multi-process debugging mode is set with the **ptrace** system call.

- 0175 (0x7D), which indicates that the process was stopped after an **exec** system call. This value only occurs when multi-process debugging mode is set with the **ptrace** system call.

If the child process stopped in a trace mode, then bits 8-15 (startin with the least significant bit) of ***stat_loc** contain the number of the signal that caused the process to stop and the low-order 8 bits are set equal to 0177.

If the child process terminated due to an **exit** system call, the low-order 8 bits of ***stat_loc** are 0 and the high-order 8 bits contain the low-order 8 bits of the parameter that the child passed to **exit** or **_exit**.

If the child process terminated due to a signal, bits 8-15 (startin with the least significant bit) of ***stat_loc** are 0 and the low-order 8 bits contain the number of the signal that caused the termination. In addition, if the low-order seventh bit (bit 0200 or 0x80) is set, then a memory image file is produced before **wait** returns.

If the Transparent Computing Facility is installed and the chil process was running on a different site, and that site left the cluster, a ***stat_loc** argument looks as though the child terminated due to a **SIGPWR** signal.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes become "orphans" and the parent process ID of each of the remaining child processes are set to -1. This is done to avoid confusion between processes which are the real child processes of the **init** process and processes which are orphaned; this is useful in a Transparent Computing Facility cluster. Note that if an orphan calls the **getppid** system call, the orphan is informed that its parent process ID is 1, **not** -1.

Note: The effect of the **wait** system call is modified by the signal action of the **SIGCHLD** signal. See "sigaction, sigvec, signal" in topic 1.2.263 for details.

Warning: The actions of the **wait** and **waitpid** system calls are undefined if the **stat_loc** parameter points to a location outside of the process's allocated address space.

Return Value

If the **wait** or **waitpid** system calls return to report the status of a suitable child process, these system calls return the value of the process ID of the child process for which the status is reported. For the **wait** system call any child process is "suitable". A "suitable" a child process for the **waitpid** system call is a member of the set of child processes that is specified by that **waitpid** system call's **pid** argument. If the **wait** or **waitpid** system calls return due to the delivery of a signal to the calling process, a value of -1 is returned and **errno** is set to the value EINTR. If a **waitpid** system call was passed an **options** argument that had WNOHANG inclusive ORed into it, and there is at least one suitable child process (of the process which called the **waitpid** system call), and status is not available for any of these suitable child processes, then that **waitpid** system call returns a value of 0. Otherwise, the **wait** and **waitpid** system calls return a value of -1, and set **errno** to indicate the error.

Error Conditions

If any of the following conditions occur, then the **wait** system call returns without waiting and returns a value of -1 and sets **errno** to the corresponding value.

ECHILD The calling process has no existing child processes not yet waited for.

EINTR The **wait** system call was interrupted by a signal. When a call to the **wait** system call indicates this error, the effect of the call on ***stat_loc** is unreliable.

If any of the following conditions occur, then the **waitpid** system call returns without waiting and returns a value of -1 and sets **errno** to the corresponding value.

ECHILD The calling process has no suitable child processes (that is, child processes that are in the set of child processes that is specified by that **waitpid** system call's **pid** argument) not yet waited for.

EINTR The **waitpid** system call was interrupted by a signal. When a call to the **waitpid** system call indicates this error, the effect of the call on ***stat_loc** is not reliable.

EINVAL The value of the **options** argument is not valid.

EFAULT If this error is detected, the **stat_loc** points to an area outside the process's allocated address space.

1.2.326 wait3

Purpose
Waits for a child process to stop or terminate.

Syntax

```
#include <sys/param.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>

pid_t wait3(status, options, rusage)
union wait *status;
int options;
struct rusage *rusage;
```

Description
The **status** and **option** words are described by definitions and macros in the file **<sys/wait.h>**; the union and its bitfield definitions and associated macros given there provide convenient and mnemonic access to the word of **status** returned by a **wait3** call. **status->w_status** is the same as the integer returned in ***stat_loc** when using the **wait** call.

If **WIFSTOPPED(*status**) is nonzero, **status->w_stopsig** is the signal which caused the child process to stop.

If **WIFSIGNALED(*status**) is nonzero, **status->w_termsig** is the signal which caused the child process to terminate and **status->w_coredump** indicates whether or not a core dump was made.

If **WIFEXITED** (***status**) is nonzero, **status->w_retcode** is the low order 8 bits of the argument that the child process passed to **exit**.

Exactly one of **WIFSTOPPED**, **WIFSIGNALED**, or **WIFEXITED** will be nonzero.

There are two **options**, which may be combined by ORing them together.

The first is **WNOHANG**, which causes the **wait3** to return immediately if there are no processes which wish to report status, returning a value of 0 in this case as the result of the **wait3**.

The second option is **WUNTRACED**, which causes **wait3** to return information about child processes of the current process which are stopped (but not traced with **ptrace**) because they received a **SIGTTIN**, **SIGTTOU**, **SIGTSTP**, or **SIGSTOP** signal. See "sigaction, sigvec, signal" in topic 1.2.263 for a description of these signals.

The **rusage** parameter may be NULL or else a pointer to an **rusage** structure, into which information describing the resources used by the process whose process-ID is returned by **wait3** (and all of its child processes for which it has collected similar information) is placed. If **rusage** is NULL, no information on resource usage is provided. Currently this information is not available for stopped processes.

If the Transparent Computing Facility is installed, the resource information returned describes the resources used on all sites by the process whose process-ID is returned by **wait3**. This may include information combined from sites of widely varying machine types. Totaling statistics from different sites, such as CPU times, may not be

appropriate.

If the child process had been running on a cluster site which appears to the parent process's site to have gone down, **wait3** reports that the child process has exited because of the SIGPWR signal. In this case, as with stopped processes, the **rusage** information is not returned.

Return Value

If **wait3** returns due to a stopped or terminated child process, the process-ID of the child process is returned to the calling process. If **WNOHANG** is specified and there are no child processes which are already stopped or terminated, a value of 0 is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions

The **wait3** system call will fail and return immediately if one or more of the following are true:

ECHILD The calling process has no existing unwaited-for child processes.

EFAULT **rusage** points to an illegal address.

EINTR The call was interrupted by a signal.

Related Information

In this book: "exit, _exit" in topic 1.2.73, " fork, vfork" in topic 1.2.83, "sigaction, sigvec, signal" in topic 1.2.263, and "wait, waitpid" in topic 1.2.325.

1.2.327 wcstring

Purpose
Performs operations on wide character strings.

Library
Standard I/O Library (**libc.a**)

Syntax

```
#include <mbcs.h>
```

```
wchar_t *wcscat (s1, s2)
wchar_t *s1;
const wchar_t *s2;

wchar_t *wcschr (s, c)
const wchar_t *s;
wchar_t c;

int wcscmp (s1, s2)
const wchar_t *s1, *s2;

wchar_t *wcscpy (s1, s2)
wchar_t *s1;
const wchar_t *s2;

size_t wcscspn (s1, s2)
const wchar_t *s1, *s2;

size_t *wcslen (s)
const wchar_t *s;

wchar_t *wcsncat (s1, s2, n)
wchar_t (s2)
const wchar_t *s2;
size_t n;

int *wcsncmp (s1, s2, n)
const wchar_t *s1, *s2;
size_t n;

wchar_t *wcsncpy (s1, s2, n).
wchar_t *s1;
const wchar_t *s2;
size_t n;

wchar_t *wcspbrk (s1, s2)
const wchar_t *s1, *s2;

wchar_t *wcsrchr (s, c)
const wchar_t *s;
wchar_t *c;

size_t wcsspn (s1, s2)
const wchar_t *s1, *s2;

wchar_t *wcstok (s1, s2)
wchar_t *s1;
const wchar_t *s2;

wchar_t *wcswcs (s1, s2)
const wchar_t *s1, *s2;
```

Description
The **wcscat** subroutine appends a copy of the wide character string pointed to by **s2** (including the terminating NULL character) to the end of the wide character string pointed to by **s2**. The initial wide character of **s2** overwrites the NULL character at the end of **s1**. If copying takes place between objects that overlap, the behavior is undefined.

The **wcschr** subroutine locates the first occurrence of **c** in the wide character string pointed to by **s**. The terminating NULL character is considered to be part of the string.

The **wcscmp** subroutine compares the wide character string pointed to by **s1** to the wide character string pointed to by **s2**. The comparison is based on the binary ordering of wide characters. This comparison is independent of the current locale.

The **wcscpy** subroutine copies the wide characters pointed to by **s2** (including the terminating NULL character) into the wide character array pointed to by **s1**. If copying takes place between objects that overlap, the behavior is undefined.

The **wcscspn** subroutine computes the length of the maximum initial segment of the wide character string pointed to by **s1**, which consists entirely of wide characters not from the string pointed to by **s2**.

The **wcslen** subroutine computes the length of a wide character string pointed to by **s**.

The **wcsncat** subroutine appends not more than **n** wide characters (a NULL character and characters that follow it are not appended) from the wide character array pointed to by **s2** to the end of the wide character string pointed to by **s1**. The initial wide characters of **s2** overwrites the NULL character at the end of **s1**. A terminating NULL character is always appended to the result. If copying takes place between objects that overlap, the behavior is undefined.

The **wcsncmp** subroutine compares not more than **n** wide characters (characters that follow a NULL are not compared) from the wide character array pointed to by **s1** to the wide character array pointed to by **s2**.

The **wcsncpy** subroutine copies not more than **n** wide characters (characters that follow a NULL character are not copied) from the wide character array pointed to by **s2** into the wide character array pointed to by **s1**. If copying takes place between objects that overlap, the behavior is undefined.

The **wcspbrk** subroutine locates the first occurrence in the string pointed to by **s1** of any wide character from the wide character string pointed to by **s2**.

The **wcsrchr** subroutine locates the last occurrence of **c** in the wide character string pointed to by **s**. The terminating NULL character is considered to be part of the string.

The **wcsspn** subroutine computes the length of the maximum initial segment of the wide character string pointed to by **s1**, which consists entirely of wide characters from the wide character string pointed to by **s2**.

The **wcstok** subroutine breaks the wide character string pointed to by **s1** into a sequence of tokens, each of which is delimited by a wide character from the wide character string pointed to by **s2**. The first call in the sequence has **s1** as its first argument, and is followed by calls with a NULL pointer as their first argument. The separator string pointed to by **s2** may be different from call to call.

The first call in the sequence searches the string pointed to by **s1** for the first character that is not contained in the current separator string pointed to by **s2**. If no such character is found, then there are no tokens in the string pointed to by **s1** and the **wcstok** subroutine returns a NULL pointer. If such a character is found, it is the start of the first token.

The **wcstok** subroutine then searches from there for a character that is contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by **s1**, and subsequent searches for a token will return a NULL pointer. If such a character is found, it is overwritten by a NULL character, which terminates the current token. The **wcstok** subroutine saves a pointer to the following character, from which the next search for a token will start.

Each subsequent call, with a NULL pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above. The saved pointer will not be changed as a side effect of calling any other library routine in **libc.a**.

The **wcswcs** subroutine locates the first occurrence in the wide character string pointed to by **s1** of the sequence of wide characters (excluding the terminating NULL character) in the wide character string pointed to by **s2**.

Return Value
The **wcscat**, **wcscpy**, **wcsncat**, and **wcsncpy** subroutines return the value of **s1**.

The **wcschr** subroutine returns a pointer to the located wide character, or a NULL pointer if the character does not occur in the string.

The **wcscmp** returns an integer greater than, equal to, or less than zero if the wide character string pointed to by *s1* is greater than, equal to or less than the wide character string pointed to by *s2*.

The **wcscspn** and **wcsspn** subroutines return the length of the segment.

The **wcslen** subroutine returns the number of wide characters that precede the terminating NULL character.

The **wcsncmp** subroutine is similar to **wcscmp**, except no more than the first *N* wide characters of *s1* or *s2* are considered in the comparison.

The **wcspbrk** subroutine returns a pointer to the first wide character, or a NULL pointer if no wide character from the wide character string from **s2** occurs in **s1**.

The **wcsrchr** subroutine returns a pointer to the wide character, or a NULL pointer if **c** does not occur in the wide character string.

The **wcstok** subroutine returns a pointer to the first wide character of the next token, or a NULL pointer if there is no token.

The **wcswcs** subroutine returns a pointer to the located wide character string, or a NULL pointer if the string is not found. If **s2** points to a wide character string with zero length, the subroutine returns **s1**.

Related Information
In this book: "mbstring" in topic 1.2.164, "string" in topic 1.2.288, and "NCstring" in topic 1.2.184.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.328 wctomb, wcstombs

Purpose
Converts wide characters and wide character strings into multibyte characters and multibyte character strings.

Syntax

```
#include <mbcs.h>

int wctomb (s, wc)
char *s;
wchar_t wc;

size_t wcstombs (s, wcs, nb)
char *s;
wchar_t *wcs;
size_t nb;
```

Description
The **wctomb** subroutine converts a wide character **wc** to a multibyte character and stores the results in **s**. This subroutine does not check for overflow of the array **s** and assumes that **s** contains enough room for the string. The string **s** will be NULL padded.

The **wcstombs** subroutine converts a wide character string **wcs** to a multibyte string and stores the results in **s**. Copying stops if a multibyte character exceeds the limit of **nb** total bytes or if a NULL character is stored.

Return Value
The **wctomb** subroutine returns:

n where **n** is the number of bytes needed to represent the multibyte character.

-1 if the value of **wc** does not correspond to a valid multibyte character.

0 if **s** is a NULL pointer or *wc* is a NULL character.

The **wcstombs** subroutine returns (**size_t**):

n where **n** is the number of bytes converted.

-1 if a value in **wcs** does not correspond to a valid multibyte character.

0 if **s** is a NULL pointer.

Related Information
In this book: "mbtowc, mbstowcs, mbstomb" in topic 1.2.165 and "setlocale" in topic 1.2.251.

The **mbcsgen** command in *AIX Operating System Commands Reference*.

AIX Guide to Multibyte Character Set (MBCS) Support.

1.2.329 wc_collate, wc_coluniq, wc_eqvmap, _wcxcol, _mbxcol, _wcxcolu, _mbxcol

Purpose
Collates characters for international character support.

Library
Standard C Library (**libc.a**)

Syntax
#include <NLchar.h>

```
long wc_collate (xc)          long wc_coluniq (xc)
wchar_t xc;                   wchar_t xc;

int_ wcxcol (index, src, xstlong wc_eqvmap (ucval)
long index;                   long ucval;
wchar_t **src, **xstr;

int_ mbxcol (index, src, xstr)
long index
wchar_t **xstr;

int_ wcxcolu (index, src, xstr)
long index;
wchar_t **src, **xstr;

int_ mbxcolu (index, src, xstr)
long index;
unsigned char **src;
wchar_t **xstr;
```

Description
AIX supports a user-configurable collating order per process, using the environment variables **LANG** or **LC_COLLATE** (see "setlocale" in topic 1.2.251). Collating values increment from 1. The **wc_collate** subroutine, called with a wide character as its argument, returns the collating value. If extended collation applies to the wide character, **wc_collate** returns a negative value and the wide character is either translated to a different character or string of characters before collation (1-to-n collation), or it collates as a unit with one or more characters following a wide character (n-to-1 collation). For example, the wide character for the code point representing "o" might translate to the string "oe" before (1-to-n) collation, or two code points representing "Pi" might translate to a unit "&pi." before (n-to-1) collation.

When **wc_collate** determines that extended collation is required, **_wcxcol** or **_mbxcol** should be called.

Like **wc_collate**, **wc_coluniq** may return a negative value which indicates that extended collation is required and that **_wcxcolu** or **_mbxcolu** should be called.

The **_wcxcol** subroutine performs extended collation on the following:

index The negative value returned from the wide character which indicates that extended collation is needed.

src A pointer to a wide character string, starting with the wide character following the one that was passed to the **wc_collate**

subroutine.

xstr A pointer to a replacement text string.

For 1-to-n collation, **_wcxcol** writes the address to **xstr** of a replacement string that is interpolated into the collating operation ahead of the remaining text of **src**.

For n-to-1 collation, a NULL value is written into the pointer.

The **_wcxcol** subroutine returns -1 if 1-to-n collation is required (**xstr** is not NULL). If n-to-1 collation is required, **_wcxcol** returns the collating value of the extended collation.

The **wc_coluniq** subroutine disables extended collation by assigning each **wchar_t** a unique value and treating it as a unit. **wc_coluniq** returns its unique collating value, a non-negative integer that does not require special interpretation. The **wc_coluniq** subroutine might be used, for example, within character ranges in regular expressions.

The **_wcxcolu** subroutine performs extended collation on the following:

index The negative value returned from **wc_coluniq**, which indicates that extended collation is needed.

src A pointer to a wide character string, starting with the wide character following the one that was passed to the **wc_coluniq** subroutine.

xstr A pointer to a replacement text string.

For 1-to-n collation, **_wcxcolu** writes the address to **xstr** of a replacement string that is interpolated into the collating operation ahead of the remaining text of **src**.

For n-to-1 collation, a NULL value is written to the pointer.

The **_wcxcolu** subroutine returns -1 if 1-to-n collation is required (**xstr** is not NULL). If n-to-1 collation is required, **_wcxcolu** returns the collating value of the extended collation.

The **wc_eqvmap** macro is a predicate that returns a non-zero value if the corresponding wide character begins an equivalence class, or a set of wide characters that can be treated as identical in some collating contexts. For example, if any character of an equivalence class is used as the beginning or ending point of a character range, all of the characters in that class are included in the range.

Related Information
In this book: "NCcollate, NCcoluniq, NCeqvmap, _NCxcol, _NLxcol" in topic 1.2.182

1.2.330 write, writex

Purpose
Writes to a file or socket.

Syntax

```
int write (d, buf, nbytes)
int d;
char *buf;
unsigned int nbytes;
```

```
int writex (d, buf, nbytes, ext)
int d;
char *buf;
unsigned int nbytes;
int ext;
```

Description
The **write** system call writes the number of bytes specified by the **nbyte** parameter from the buffer pointed to by the **buf** parameter to the object associated with the **d** parameter.

The **d** parameter is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call, or a socket descriptor from a **socket** or **socketpair** system call.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from the **write** system call, the file pointer increments by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

When the **O_APPEND** flag of the file status is set, the file pointer is set to the end of the file prior to each write.

If a file is opened with the **O_DEFERC** flag, changes made to it with **write** can later be undone with the **fabort** system call. Changes are not made permanent until the file is committed explicitly by calling **fsync** or implicitly some time after calling **close** or exiting, at which time all files are closed. (See "fabort" in topic 1.2.75, "fsync, fcommit" in topic 1.2.87, and "close, closex" in topic 1.2.48.)

If this file is open for writing using more than one file descriptor, an **fabort** operation will undo uncommitted changes made to the file by any of these processes. Also, the implicit commit of the changes will not occur until all of these file descriptors are closed. If one wants to guarantee that the changes are made permanent immediately, the use of **fcommit** is recommended. Also, locking such as that provided by the **lockf** system call is recommended if more than one process will be writing to the same file.

If the real user is not the superuser, the **write** system call clears the **set-group-id** and **set-user-id** bits on a file. This prevents penetration of system security by a user who obtains a writable **set-user-id** file owned by the superuser.

If the **write** system call requests that more bytes be written than there is room for, only as many bytes as there is room for are written and the **write** system call returns an integer equal to the number of bytes written. The next attempt to write a nonzero number of bytes will fail (except as noted following). The limit reached can be either the ulimit (see "ulimit" in topic 1.2.313) or the end of the physical medium. A partial

write is not permitted for the following:

If the file being written is a pipe (or FIFO) and unless **O_NDELAY** or **O_NONBLOCK** is set in the file flag word, a **write** to a full pipe (or FIFO) blocks until space becomes available.

If the file being written is a pipe (or FIFO) and **O_NDELAY** is set in the file flag word, a **write** to a full pipe (or FIFO) returns a count of 0.

If the file being written is a pipe (or FIFO) and **O_NONBLOCK** is set in the file flag word, a **write** to a full pipe (or FIFO) returns -1 and **errno** is set to EAGAIN.

An attempt to write data to a pipe when the read end of the pipe ha been closed will cause the signal **SIGPIPE** to be raised. If this signal is caught or ignored, the error EPIPE is returned.

When attempting to write to a socket and the socket is not ready to accept data:

Unless **O_NDELAY** or **O_NONBLOCK** is set, the **write** blocks until the socket is ready to accept data.

If **O_NDELAY** is set, the **write** returns 0.

If **O_NONBLOCK** is set, the **write** returns -1 and **errno** is set to EAGAIN.

If the file to be written supports enforcement mode record locks and all or part of the region to be written is currently locked by another process:

If **O_NDELAY** or **O_NONBLOCK** is set, **write** returns -1 and sets **errno** to EAGAIN.

If **O_NDELAY** and **O_NONBLOCK** are not set, the calling process blocks until the lock is released.

For more information about record locks, see "fcntl, flock, lockf" in topic 1.2.78 .

The **SIGTTOU** signal will be sent to the writer's process group if the process:

is writing to its controlling termina

has the TOSTOP terminal attribute se

is not ignoring or blocking **SIGTTOU**, and

is in the foreground process group of the termina

Otherwise, output to a terminal is permitted, even by processes which are not in the foreground process group of the terminal.

If the Transparent Computing Facility is installed on your system, when writing to a file which is stored on a site other than where the process is running, it is possible for the out-of-space condition to go unnoticed until after the **write** system call has returned. If this situation arises, the ENOSPC error will be reported on subsequent **write**, **fcommit**, and **fabort**

calls. The system attempts to minimize the likelihood of out-of-space problems getting reported in this way by using a less efficient, synchronous procedure for writing to a remote file system when that file system is low on space. Thus, writing to a remote file system is most efficient when the file system is less than 90% full.

The **writex** system call performs the same function as **write**, except that it provides communication with character device drivers that require more information or return more status than **write** can handle.

For files, directories, sockets, or special files with drivers that do not handle extended operations, the **writex** system call does exactly what the **write** system call does, and the **ext** parameter is ignored.

Each driver interprets the **ext** parameter in a device-dependent way, either as a value or as a pointer to a communication area. The nonextended **write** system call is equivalent to the extended **writex** system call with an **ext** parameter value of 0. Drivers must apply reasonable defaults when the **ext** parameter value is 0.

Return Value
Upon successful completion, the number of bytes actually written is returned. If the **write** or **writex** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Error Conditions
The **write** and **writex** system calls fail and the file pointer remains unchanged if one or more of the following are true:

EBADF The **d** parameter is not a valid file descriptor open for writing or valid socket descriptor.

EAGAIN An enforcement mode record lock is outstanding in the portion of the file that is to be written.

EAGAIN The object is a socket or pipe and is marked for non-blocking I/O, and the socket or pipe was not ready to accept data.

EPIPE An attempt is made to write to a pipe that is not open for reading by any process or to write to a socket of type **SOCK_STREAM** that is not connected to a peer socket. A **SIGPIPE** signal is also sent to the calling process.

EFBIG An attempt is made to write a file that exceeds the process's file size limit or the maximum file size (see "ulimit" in topic 1.2.313).

EFAULT **buf** points to a location outside of the process's allocated address space.

EDEADLK A deadlock would occur if the calling process were to sleep until the region to be written was unlocked.

EINTR A signal was caught during the **write** system call.

ENOSPC There is no more space left on the device.

EIO A physical I/O error occurred.

If the Transparent Computing Facility is installed on your system, **write**

or **writex** can also fail if one or more of the following are true:

ESITEDN1 The file cannot be written because the file storage site went down. If the file is replicated, this indicates the loss of the site where the primary copy of the file is stored.

ESITEDN2 The operation was terminated because a site failed.

Related Information

In this book: "dup" in topic 1.2.64, "fcntl, flock, lockf" in topic 1.2.78, "lseek" in topic 1.2.161, "open, openx, creat" in topic 1.2.199, "pipe" in topic 1.2.204, "ulimit" in topic 1.2.313, and "writev" in topic 1.2.331.

1.2.331 writev

Purpose
Writes output gathered from multiple buffers.

Syntax

#include <sys/uio.h>

int writev (d, iov, iovcnt)
int d;
struct iovec *iov;
int iovcnt;

Description
The **writev** system call attempts to write the data in the buffers specified by the array of **iovec** structures, pointed to by the **iov** parameter, to the object associated with the **d** parameter.

The **d** parameter is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call, or a socket descriptor from a **socket** or **socketpair** system call.

The **iovec** structure is defined in the **sys/uio.h** header file, and it contains the following members:

```
caddr_t  iov_base;
int      iov_len;
```

Each **iovec** entry specifies the base address and length of an area in memory from which data should be written. The **writev** system call always writes a complete area before moving to the next.

On devices capable of seeking, **writev** starts at a position in the file given by the file pointer associated with the **d** parameter. Upon return from the **writev** system call, the file pointer is incremented by the number of bytes actually written.

Devices that are incapable of seeking always write at the current position. The value of a file pointer associated with such a file is undefined.

When the **O_APPEND** flag of the file status is set, the file pointer is set to the end of the file prior to each write.

If a file is opened with the **O_DEFERC** flag, changes made to it with **writev** can later be undone with the **fabort** system call. Changes are not made permanent until the file is committed explicitly by calling **fsync** or implicitly some time after calling **close** or exiting, at which time all files are closed. (See "fabort" in topic 1.2.75, "fsync, fcommit" in topic 1.2.87, and "close, closex" in topic 1.2.48.)

If the file is opened with the **O_SYNC** or **O_REPLSYNC** flags, changes are made permanent and possibly replicated. (See "fcntl, flock, lockf" in topic 1.2.78, "open, openx, creat" in topic 1.2.199, and "write, writex" in topic 1.2.330.)

If the **writev** system call requests that more bytes be written than there is room for, only as many bytes as there is room for are written and the **writev** system call returns an integer equal to the number of bytes

written. The next attempt to write nonzero number of bytes will fail (except as noted following). The limit reached can be either the ulimit (see "ulimit" in topic 1.2.313) or the end of the physical medium. A partial write is not permitted for the following:

If the file being written is a pipe (or FIFO) and unless **O_NDELAY** or **O_NONBLOCK** is set in the file flag word, then a **writev** to a full pipe (or FIFO) blocks until space becomes available.

If the file being written is a pipe (or FIFO) and **O_NDELAY** is set in the file flag word, then a **writev** to a full pipe (or FIFO) returns a count of 0.

If the file being written is a pipe (or FIFO) and **O_NONBLOCK** is set in the file flag word, then a **writev** to a full pipe (or FIFO) returns -1 and **errno** is set to EAGAIN.

When attempting to write to a socket and the socket is not ready to accept data:

Unless **O_NDELAY** or **O_NONBLOCK** is set, the **writev** blocks until the socket is ready to accept data.

If **O_NDELAY** is set, the **writev** returns 0.

If **O_NONBLOCK** is set, the **writev** returns -1 and **errno** is set to EAGAIN.

If the file to be written supports enforcement mode record locks and all or part of the region to be written is currently locked by another process:

If **O_NDELAY** or **O_NONBLOCK** is set, then **writev** returns -1 and sets **errno** to EAGAIN.

If **O_NDELAY** and **O_NONBLOCK** are not set, then the calling process blocks until the lock is released.

For more information about record locks, see "fcntl, flock, lockf" in topic 1.2.78.

If the Transparent Computing Facility is installed on your system, when writing to a file which is stored on a site other than where the process is running, it is possible for the out-of-space condition to go unnoticed until after the write system call has returned. If this situation arises, the **ENOSPC** error will be reported on subsequent write, fcommit, and fabort calls. The system attempts to minimize the likelihood of out-of-space problems getting reported in this way by using a less efficient, synchronous procedure for writing to a remote file system when that file system is low on space. Thus, writing to a remote file system is most efficient when the file system is less than 90% full.

Return Value

Upon successful completion, **writev** returns the number of bytes that were actually written. Otherwise, the value -1 is returned and **errno** is set to indicate the error.

Error Conditions

The **writev** system call fails when one or more of the following is true:

EBADF The **d** parameter is not a valid file descriptor open for writing or a valid socket descriptor.

EAGAIN An enforcement mode record lock is outstanding in the portion of the file that is to be written.

EAGAIN The object is a socket or pipe and is marked for nonblocking I/O, and the socket or pipe was not ready to accept data.

EPIPE An attempt is made to write to a pipe that is not open for reading by any process or to write to a socket of type **SOCK_STREAM** that is not connected to a peer socket. A **SIGPIPE** signal is also sent to the calling process.

EFBIG An attempt is made to write to a file that exceeds the process's file size limit or the maximum file size (see "ulimit" in topic 1.2.313).

EINVAL The value of **iovcnt** was not between 1 and 16, inclusive.

EINVAL One of the **iov_len** values in the **iov** array was negative.

EINVAL The sum of the **iov_len** values in the **iov** array overflowed a 32-bit integer.

EFAULT Part of the **iov** points to a location outside of the process's allocated address space.

EDEADLK A deadlock would occur if the calling process were to sleep until the region to be written was unlocked.

EINTR A signal was caught during the **writev** system call.

ENODEV The file specified is an invalid device for writing.

EIO A physical I/O error occurred.

If the Transparent Computing Facility is installed on your system, **writev** can also fail if one or more of the following are true:

ESITEDN1 The file cannot be written because the file storage site went down. If the file is replicated, this indicates the loss of the site where the primary copy of the file is stored.

ESITEDN2 The operation was terminated because a site failed.

Related Information

In this book: "fcntl, flock, lockf" in topic 1.2.78, "lseek" in topic 1.2.161, "open, openx, creat" in topic 1.2.199, "pipe" in topic 1.2.204, "select" in topic 1.2.242, and "write, writex" in topic 1.2.330.

1.2.332 XDR (External Data Representation)

Purpose
Allows programmers to describe basic data types in a uniform representation.

Library
Internet Library (**libc.a**)

Syntax
#include <rpc/rpc.h>

Description
XDR provides programmers with a specification of uniform representations for basic data types. XDR does not depend on machine languages, operating systems, or architectures; thus, networked computers can share data regardless of the machine on which the data is produced or consumed.

For basic data types such as integers and strings, XDR provides primitives that serialize, or translate, information from the local host's representation to XDR representation, and deserialize, or translate, from the XDR representation to the local host's representation. XDR also uses constructor primitives that allow the use of the basic data types to create more complex data types, such as arrays and discriminated unions.

XDR describes input and output data structures in a data description language that resembles the C programming language. It is important to remember that C language constructs define the code for programs while XDR standardizes the representation of data types in the programming code. Representing data in standardized formats resolves situations that can occur if different byte ordering exists on networked machines and enables machines with different structure alignment algorithms to communicate with each other.

The XDR routines are not dependent on direction. That is, the same routine is called to serialize and deserialize data. To achieve this independence, XDR passes the addresses of the objects instead of passing the object itself.

XDR is based on the assumption that bytes (or 8 bits of data, which are also called an octet) can be ported and encoded on media that can preserve the meaning of the bytes across the hardware boundaries of data. For example, bytes are ported and encoded from low order to high order in local area networks.

XDR does not represent bit fields or bit maps. It represents data in blocks of multiples of 4 bytes (32 bits). The bytes are numbered from 0 (zero) to the value of n-1, where the value (**n** mod 4) = 0. They are read or written to a byte stream in the order that byte **m** precedes byte **m+1**.

Subtopics
1.2.332.1 XDR Subroutines

1.2.332.1 XDR Subroutines

The XDR subroutines discussed in this section cover the following:

Library primitives for basic data types and constructed data types Basic data types include the number filters (for integers, floating-point and double-precision numbers), enumeration filters, and the routine for passing no data. Constructed data types include the filters for strings, arrays, unions, pointers, and opaque data.

Data stream creation routines that are used to call streams fo serializing (and deserializing) data to or from standard I/O file streams, TCP/IP connections, AIX files, and memory.

Implementation of new XDR streams

Passing linked lists using XDR

Before the XDR routines are discussed in detail, XDR data type representations are introduced and defined in the next section to make the discussion of the routines easier to understand.

Subtopics

1.2.332.1.1 XDR Data Type Representation

While many of XDR's data representations are similar to the data types used in C language constructs, there are some differences. For that reason, the data objects converted to XDR representation are briefly described in this section. More detailed information about data types can be found in *IBM AIX C Language Guide and Reference*

Integers: An XDR integer is a numerical integer value in the range [-2,147,483,648 to 2,147,483,647] that occupies 32 bits. An integer is represented in two's complement notation. Its most significant byte is 0 and least significant byte is 3.

Unsigned Integers: An XDR integer is a numerical integer value that encodes a nonnegative integer from the range [0 to 4294967295] that occupies 32 bits. It is represented by an unsigned binary number whose most significant byte is 0 and least significant byte is 3. The shortened form is **u_**.

Enumerations: Enumerations have the same representation as integers. They are used to describe subsets of integers. Enumerations are defined as follows:

```
typedef enum { name = value, ... } type-name;
```

For example, the colors red, yellow, and blue are described in an enumerated type by the following:

```
typedef enum { RED = 2, YELLOW = 3, BLUE = 5 } colors;
```

Booleans: A Boolean is an enumeration that takes the following form:

```
typedef enum { FALSE = 0, TRUE = 1 } boolean;
```

Floating-Point Data: An XDR floating point is data that encodes normalized single floating-point numbers that conform to IEEE standard and occupy 32 bits (4 bytes). A floating-point number is made up of an integer with an exponent. The integer can contain a fraction value. Its most significant bit is 0 and least significant bit is 31. The representation for floating points is the following:

```
(-1)(S*)2(E-Bias*)1.F
```

The following describes the IEEE coding standard for the fields represented in this notation:

S The sign of the number. The value 0 specifies positive and the value 1, negative. The most significant bit is 0.

E Exponent of the number in base 2. Floats devote 8 bits to this field. It is biased by 127. The most significant bit is 1.

F Fractional part of the number's mantissa in base 2. Floats devote 23 bits to this field. The most significant bit is 9.

Note: Consult the IEEE specification when encoding signed 0, signed infinity (overflow), and denormalized numbers (underflow). Under IEEE specifications, the value of **NaN** depends on your operating system. **NaN** represents the phrase *not* a number. It is not recommended for describing data types.

Double-Precision Data: XDR double-precision data encodes double-precision, floating-point numbers that conform to IEEE standard and occupy 64 bits (8 bytes). A double-precision point is made up of an integer with an exponent. The integer can contain a fractional value. The most significant bit is 0 and least significant bit is 63. Double-precisions can be represented in this following form:

(-1)(S*)2(E-Bias*)1.**F**

The following describes the IEEE coding standard for the fields represented in this notation:

S The sign of the number. The value 0 specifies positive and the value 1, negative. The most significant bit is 0.

E Exponent of the number in base 2. Doubles devote 11 bits to this field. It is biased by 1023. The most significant bit is 1.

F Fractional part of the number's mantissa in base 2. Doubles devote 52 bits to this field. The most significant bit is 12.

Note: Consult the IEEE specification when encoding signed zero, signed infinity (overflow), and denormalized numbers (underflow). Under IEEE specifications, the value of **NaN** depends on your operating system. **NaN** represents the phrase **not** a number. It is not recommended for describing data types.

Opaque Data: XDR opaque data is data of a fixed size that is passed to another machine without being interpreted. It is encoded in multiples of 4 bytes and defined by the following:

```
typedef opaque type-name[n];
opaque name[n];
```

In the definition, **n** is the static number of bytes needed to contain the opaque data. If **n** is not a multiple of four, the **n** bytes are followed by the number of bytes needed to make the opaque data object's total byte count a multiple of four.

Counted Byte Strings: XDR-counted byte strings are strings of **n** bytes (numbered 0 through n-1) encoded as an unsigned integer followed by the actual bytes of the string. For example, the counted byte string **examples** is encoded as **8examples** (8 bytes, and then the actual bytes for the word **examples**).

If **n** is not a multiple of four, the **n** bytes are followed by the number of bytes needed to make the opaque data object's total byte count a multiple of four.

The data description of strings follows:

```
typedef string type-name<n>;
typedef string type-name<>;
string name <n>;
string name<>;
```

The XDR data description language uses **<** and **>** (angle brackets) for objects with variable lengths, and **[** and **]** (square brackets) for objects

with fixed lengths.

The constant **n** specifies the maximum number of bytes a string can contain. If **n** is not specified, XDR assumes the maximum length is 2(32)-1. The constant **n** is listed in a protocol specification. For example, if a protocol specifies that a file name can be no longer than 255 bytes, it is represented as **string filename<255>;**.

Fixed Arrays: XDR fixed arrays are sets of homogenous, or identical, elements of fixed sizes. The array's elements are encoded in their natural order of 0 to **n**-1. The data description for fixed-size arrays of homogeneous elements is defined as follows:

```
typedef elementtype type-name[n];
elementtype name[n];
```

Counted Arrays: XDR-counted arrays are sets of homogenous, or identical, elements of varying lengths. The array is encoded starting with the element count specified by an unsigned integer (**n**) followed by each array element from 0 through **n**-1.

The data description for counted arrays follows:

```
typedef elementtype type-name<n>;
typedef elementtype type-name<>;
elementtype name<n>;
elementtype name<>;
```

The constant **n** specifies the maximum number of an element count of an array. If **n** is not specified, XDR assumes the maximum length is 2(32)-1. The constant **n** is listed in a protocol.

Structures: XDR structures are sets of components put together to create a specific data set. They are very similar to the standard C language structures.

A structure's components are encoded in the order of their declaration in the structure. The data description for structures follows:

```
typedef struct {
        component-type component-name;
        ...
} type-name;
```

Discriminated Unions: An XDR discriminated union is a set of data composed of a discriminant and another data type. The discriminant is an enumeration. The other data type is selected from a set of prearranged types according to the value of the discriminant. The component types are called arms of the union. The discriminated union is encoded starting with the discriminant followed by the arm.

The data description for discriminated unions is as follows:

```
typedef union switch (discriminant-type) {
        discriminant-value: arm-type;
        ...
        default: default-arm-type;
} type-name;
```

Note: The default arm is optional. If it is not specified, a union

cannot encode discriminant values that are not specified. Most specifications do not use default arms.

1.2.332.1.2 XDR Library Routines

XDR routines are organized as a library of primitives that define the data types as well as create data streams. In this section, the routines are grouped by function. The filter primitives for basic and constructed data types appear first, followed by the non-filter primitives for manipulating XDR streams.

When using XDR with RPC (Remote Procedure Call), it is important to note that RPC clients do not create data streams themselves. The RPC system creates the streams itself. RPC passes the information about the data streams as opaque data in the form of handles. This opaque data handle is referred to in routines as the **xdrs** parameter.

Programmers who use C language programs with XDR routines must include the **<rpc/xdr.h>** file, which contains the necessary XDR interfaces.

Since XDR allows programmers to read and write C language constructs, programmers can also write their own XDR routines to define other data types.

For each data type, there is an XDR routine associated with it. These XDR routines take the following form:

```
xdr_xxx (xdrs, fp)
        XDR *xdrs;
        xxx *fp;
{
}
```

The **xxx** parameter represents a data type. The **xdrs** parameter is an opaque handle that points to an XDR stream. The opaque handle pointer is passed to the primitive XDR routines. The **fp** parameter points to an address of a data value that provides data to the stream or receives data from it.

Unless noted otherwise, XDR routines return the value 1 if they succeed and the value 0 if they do not.

1.2.332.1.3 Filter Primitives

Filter primitives define basic and constructed data types. Basic data types include numbers, floating points, and generic enumerations. Constructed data types include strings, byte arrays, opaque data, pointers and unions. Constructed data type primitives require more parameters and perform more complicated functions, such as memory management.

Basic Data Types: The XDR basic data types include number filters, floating-point filters, and enumeration filters. Also included with these primitives is a routine for use when no data is exchanged.

Number Filters

The XDR library provides primitives that translate between types of numbers and external representations. The XDR number filters cover the signed and unsigned integers as well as the signed and unsigned short and long integers. The following list of routines are the XDR library number filters.

xdr_int (xdrs, ip)
XDR *xdrs;
int *ip;

The **xdr_int** filter primitive translates between C language integers and their external representations. The **xdrs** parameter is an XDR stream handle. The **ip** parameter points to the address of the number that provides data to the XDR stream or receives data from it.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

xdr_u_int (xdrs, up)
XDR *xdrs;
u_int *up;

The **xdr_u_int** filter primitive translates between C language unsigned integers and their external representations. The **xdrs** parameter is an XDR stream handle. The **up** parameter points to the address of the number that provides data to the XDR stream or receives data from it.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

xdr_long (xdrs, lip)
XDR *xdrs;
long *lip;

The **xdr_long** filter primitive translates between C language long integers and their external representations. The **xdrs** parameter is an XDR stream handle. The **lip** parameter points to the address of the number that provides data to the XDR stream or receives data from it.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

xdr_u_long (xdrs, lup)
XDR *xdrs;

u_long *lup;

The **xdr_u_long** filter primitive translates between C language unsigned long integers and their external representations. The **xdrs** parameter is an XDR stream handle. The **lup** parameter points to the address of the number that provides data to the XDR stream or receives data from it.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

xdr_short (xdrs, sip)
XDR *xdrs;
short *sip;

The **xdr_short** filter primitive translates between C language short integers and their external representations. The **xdrs** parameter is an XDR stream handle. The **sip** parameter points to the address of the number that provides data to the XDR stream or receives data from it.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

xdr_u_short (xdrs, sup)
XDR *xdrs;
u_short *sup;

The **xdr_u_short** filter primitive translates between C language unsigned short integers and their external representations. The **xdrs** parameter is an XDR stream handle. The **sup** parameter points to the address of the number that provides data to the XDR stream or receives data from it.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

Floating-Point Filters

The XDR library provides primitives that translate between floating-point data and their external representations. Floating-point data encodes an integer with an exponent. Floats and double-precision numbers compose floating-point data.

Note: Numbers are represented as IEEE standard floating points. Routines may fail when decoding IEEE representations into machine-specific representations or vice-versa.

xdr_float (xdrs, fp)
XDR *xdrs;
float *fp;

The **xdr_float** filter translates between C floats (normalized single floating-point numbers) and their external representations. The **xdrs** parameter is an XDR stream handle. The **fp** parameter points to the address of the float that provides data to the XDR stream or receives data from it.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

xdr_double (xdrs, dp)
XDR *xdrs;
double *dp;

The **xdr_double** filter translates between C double-precision numbers and their external representations. The **xdrs** parameter is an XDR stream handle. The **dp** parameter points to the address of the double-precision number that provides data to the XDR stream or receives data from it.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

Enumeration Filters

The XDR library provides a primitive for generic enumerations that is based on the assumption that a C enumeration value (**enum**) has the same representation. There is a special enumeration in XDR known as Boolean.

xdr_enum (xdrs, ep)
XDR *xdrs;
enum_t *ep;

The **enum** filter primitive translates between C language **enums**, which are actually integers, and their external representations. The **xdrs** parameter is an XDR stream handle. The **ep** parameter points to the address of the enumeration data that provides data to the XDR stream or receives data from it.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

xdr_bool (xdrs, bp)
XDR *xdrs;
bool_t *bp;

The **bool** filter primitive translates between Booleans and their external representations which is either 1 (for TRUE) or 0 (for FALSE). The **xdrs** parameter is an XDR stream handle. The **bp** parameter points to the address of the Boolean data that provides data to the XDR stream or receives data from it.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

No Data

From time to time, an XDR routine must be supplied to the RPC system, but no data is required or passed. The XDR library provides the following primitive for this function:

xdr_void ()

The **xdr_void** primitive has no function parameters, and the routine always returns the value 1.

Constructed Data Type Filters: Constructed data type filters allow complex data types to be created from basic data types. They require more parameters in order to perform more complicated functions than basic data

types. The constructed data types can use memory management. Memory is allocated when deserializing data with the **XDR_DECODE** subroutines. Memory is deallocated through the **XDR_FREE** subroutines.

Strings

A string is defined as a sequence of bytes terminated by a null byte. The null byte does not figure into the length of the string. Externally, strings are represented by a sequence of ASCII characters. Internally, XDR represents them as pointers to characters. The XDR library includes primitives for strings and wrap strings.

xdr_string (xdrs, sp, maxlength)
XDR *xdrs;
char **sp;
u_int maxlength;

The **xdr_string** filter primitive translates between strings and their external representation. The **xdrs** parameter points to the XDR stream handle. The **sp** parameter points to the address of the pointer to the string. The **maxlength** parameter specifies the maximum length of the string allowed during encoding or decoding. This value is set in a protocol. For example, if a protocol specification specifies that a file name cannot be longer than 255 characters, a string cannot exceed 255 characters.

The routine returns 0 if the number of characters exceeds **maxlength**, or 1 if it does not.

When serializing a string, the **sp** parameter points to a string of a certain length. If the string does not exceed **maxlength**, the bytes are serialized.

When deserializing a string, the length of the incoming string is determined. The string must not exceed **maxlength**. Next, **sp** is dereferenced. If the value of ***sp** is NULL, a string of the appropriate length is allocated and ***sp** is set to this string. If the original value of ***sp** is not NULL, XDR assumes that a target area has been allocated to hold the strings that are no longer than specified by **maxlength**. The string is decoded into the target area, and the routine appends a null character to the string.

In the **XDR_FREE** subroutine, the string is obtained by dereferencing **sp**. If the value of the string is not NULL, the data allocated to the string is freed and the pointer is set to NULL.

xdr_wrapstring (xdrs, sp)
XDR *xdrs
char **sp

The **xdr_wrapstring** filter primitive calls the **xdr_string** routine with the maximum length set as the maximum value of an unsigned integer. This is a useful routine since it passes only two parameters instead of the three required by **xdr_string**. The **xdrs** parameter points to the XDR stream handle. The **sp** parameter points to the address of the pointer to the string.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

Byte Arrays

The XDR library provides a primitive for byte arrays. Although similar to strings, byte arrays differ from strings by having a byte count. That is, the length of the array is set by an unsigned integer. They also differ since byte arrays are not terminated with a null character. External and internal representation of byte arrays are the same.

```
xdr_bytes (xdrs, bpp, lp, maxlength)
XDR *xdrs;
char **bpp;
u_int *lp;
u_int maxlength;
```

The **xdr_bytes** filter primitive translates between counted byte strings and their external representation. The **xdr_bytes** primitive handles a subset of generic arrays, in which the size of the element is 1, and each element's external description is built-in.

The **xdrs** parameter points to the XDR stream handle. The **bpp** parameter points to the address of the pointer to the byte array. The **lp** parameter points to the length of the byte area. When serializing, XDR gets the length of the byte area by dereferencing **lp**. When deserializing, ***lp** is set to the byte length. The **maxlength** parameter specifies the maximum number of bytes allowed when XDR encodes or decodes messages.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

Arrays

The XDR library provides a filter primitive for handling arrays of arbitrary elements. Arrays of this type are handled in much the same way as a byte array, which handles a subset of generic arrays where the size of the elements and their external description is predetermined. This primitive for generic arrays elements requires an additional parameter to define the size of the array and to call an XDR routine to encode or decode each element in the array.

```
xdr_array (xdrs, arrp, sizep,maxlength, elsize, elproc)
XDR *xdrs;
char **arrp;
u_int *sizep;
u_int maxlength;
u_int elsize;
xdrproc_t elproc;
```

The **xdr_array** filter primitive translates between sets of arbitrary elements and their corresponding external representations.

The **xdrs** parameter points to the XDR stream handle. The **arrp** parameter points to the address of the pointer to the array. The **sizep** parameter points to the address of the element count of the array. The **maxlength** parameter specifies the maximum number of array elements. The **elsize** parameter specifies the size in bytes of each of the array's elements.

The **elproc** parameter specifies the name of the XDR routine called to serialize, deserialize, or free each element in the array.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

Fixed-Length Arrays

XDR does not provide a primitive for defining a fixed-length array. Programmers can set the size of fixed length arrays by including conditional statements in the syntax of an **xdr_array** routine.

The following examples show how to construct a fixed-length array. First, a new data structure is defined, and then an XDR routine that serializes (and deserializes) the new structure is coded. The following data structure defines a network user as **netuser**:

```
struct netuser {
     char  *nu_machinename;
     int   nu_uid;
     u_int nu_glen;
     int   nu_gids;
};
#define NLEN 255
#define NGRPS 20
```

The **nu_machinename** parameter specifies the network user's computer host name, which can be obtained by using the **gethostname** command. The **nu_uid** parameter specifies the network user's user ID, which can be obtained by using the **geteuid** command. The **glen** parameter specifies the length of the group array while the **nu_gids** parameter specifies the network user's group IDs, which can be obtained by using the **getgroups** command. **NLEN** sets the local host name length maximum at 255 characters. **NGRPS** sets the maximum number of groups the user can be in at 20 groups.

The XDR routine generated to serialize and deserialize the **netuser** structure is coded as follows:

```
bool_t
xdr_netuser (xdrs, nup)
     XDR *xdrs;
     struct netuser *nup;
{
     return (xdr_string(xdrs,&     nup->     nu_machinename, NLEN) &&
       xdr_int (xdrs, &       nup->nu_uid) &&
       xdr_array (xdrs,&        nup->nu_gids,&        nup->nu_glen, NGRPS,
       sizeof (int), xdr_int));
}
```

The **xdrs** parameter identifies an XDR stream handle. The **nup** parameter points to the address of the structure **netuser**.

To code a routine to use fixed-size arrays, the preceding example can be rewritten as follows:

```
#define NLEN 255
#define NGRPS 20

struct netuser {
     char  *nu_machinename;
```

```
        int   nu_uid;
        int   nu_gids;
  };

  bool_t
  xdr_netuser (xdrs, nup)
        XDR *xdrs;
        struct netuser *nup;
  {
        int i;

        if (!xdr_string(xdrs;&     nup->     nu_machinename, NLEN))
           return (FALSE);
        if (!xdr_int (xdrs, &     nup->nu_uid))
           return (FALSE);
        for (i = 0; i < NGRPS; i++)  {
           if (!xdr_int (xdrs, &        nup->nu_gids        [i]))
               return (FALSE);
        }
        return (TRUE);
  }
```

This program sets the size of the fixed-array using the parameters as specified in the preceding examples.

Opaque Data

Opaque data is composed of bytes of a fixed size that are not interpreted as they pass through the data streams. Opaque data bytes, such as handles, are passed back and forth from the servers to clients without being inspected by the client. The client uses the data as it is and then returns it to the server. By definition, the actual data contained in the opaque object is not portable between computers.

```
bool_t
xdr_opaque (xdrs, p, len)
XDR *xdrs;
char *p;
u_int len;
```

The **xdr_opaque** filter primitive translates between opaque data and its external representation. The **xdrs** parameter points to the XDR stream handle. The **p** parameter points to the address of the opaque object while the **len** parameter specifies the size, in bytes, of the object.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

Discriminated Unions

A discriminated union is a C language union, which is an object that holds several data types, where one arm of the union is an enumeration value, or discriminant, that holds a specific object that is processed over the system first. The discriminant is an enumeration value (**enum_t**).

```
xdr_union (xdrs, dscmp, unp, armchoices, defaultarm)
XDR *xdrs;
enum_t *dscmp;
char *unp;
```

```
struct xdr_discrim *armchoices;
xdrproc_t defaultarm;
```

The **xdr_union** filter primitive translates between discriminated unions and their external representations.

The **xdrs** parameter points to the XDR stream handle. The **dscmp** parameter points to the address of the union's discriminant. The discriminant is an **enum_t** value. The **unp** parameter is a character pointer to the address of the union. The **armchoices** parameter points to an array of structures that define other data types. The **defaultarm** parameter is a structure provided in case no discriminants are found.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

Pointers to Structures

The XDR library provides the primitive for pointers so that structures referenced within other structures can be serialized, deserialized, and freed without causing problems. The **xdr_reference** primitive cannot attach special meaning to a NULL pointer during serialization. Attempting to pass the address of a NULL pointer can cause a memory fault. Programmers must be sure to describe data with a two-armed discriminated union. One arm is used when the pointer is valid. The other is used when the pointer is NULL.

```
xdr_reference (xdrs, pp, size, proc)
XDR *xdrs;
char **pp;
u_int size;
xdrproc_t proc;
```

The **xdr_reference** filter primitive provides chase pointers to structures within structures.

The **xdrs** parameter points to the XDR stream handle. The **pp** parameter points to the address of the pointer to the structure. When decoding data, XDR allocates storage if the pointer is NULL. The **size** parameter specifies the size of the structure pointed to. The **proc** parameter specifies the XDR procedure that describes the structure.

Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

1.2.332.1.4 Non-Filter Primitives

The XDR non-filter primitives are used to manipulate XDR streams.

u_int
xdr_getpos (xdrs)
XDR *xdrs;

The **xdr_getpos** routine returns an unsigned integer that describes the current position in the data stream. The **xdrs** parameter points to the XDR stream handle.

In some XDR streams, this routine returns the value -1 even if the value has no meaning.

bool_t
xdr_setpos (xdrs, pos)
XDR *xdrs;
u_int pos;

The **xdr_setpos** primitive changes the current position in the XDR stream. The **xdrs** parameter points to the XDR stream handle. The **pos** parameter is the new position setting.

Upon successfully repositioning the stream, the routine returns the value 1. In some XDR streams, you cannot set a position. If you try to set a position in one of these XDR streams, the routine fails and the value 0 is returned. This routine also fails if you request a position that is not within the stream's boundaries. (Boundaries vary across streams.)

void
xdr_destroy (xdrs)
XDR *xdrs;

The **xdrs_destroy** routine destroys the XDR stream pointed to by the **xdrs** parameter, and frees the private data structures allocated to the stream. The use of the XDR stream handle is undefined after it is destroyed.

1.2.332.1.5 XDR Operation Directions

The value represented by **xdrs->x_op** is either **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**. These operation values are handled internally by the XDR routines themselves. This means the same XDR routine can be called to serialize and deserialize data.

1.2.332.1.6 Data Stream Access

XDR data streams are obtained by calling creation routines. These creation routines take arguments specifically designed to the properties of the stream. There are existing XDR data streams for serializing (and deserializing) data in standard input/output files, memory, and TCP/IP connections and files.

Note: RPC clients do not need to create XDR streams. The RPC system creates them and passes them to the clients.

1.2.332.1.7 Standard I/O Streams

XDR data streams serialize and deserialize standard input/output by calling the standard input/output creation routine, **xdrstdio_create**.

```
#include <stdio.h>
#include <rpc/rpc.h>

void
xdrstdio_create (xdrs, file, x_op)
XDR *xdrs;
FILE *file;
enum xdr_op x_op;
```

The **xdrstdio_create** routine initializes the XDR data stream pointed to by the **xdrs** parameter. The **file** parameter points to the standard input/output device from which data is written or read.

The **x_op** parameter specifies an XDR direction. The possible choices are **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**.

1.2.332.1.8 Memory Streams

XDR data streams serialize and deserialize data from memory by calling the XDR memory creation routine, **xdrmem_create**. In RPC, UDP/IP implementation of remote procedure calls use this routine to build complete call and reply messages in memory before sending the message to the recipient.

```
#include <rpc/rpc.h>

void
xdrmem_create (xdrs, addr, len, x_op)
XDR *xdrs;
char *addr;
u_int len;
enum xdr_op x_op;
```

The **xdrmem_create** routine initializes in local memory the XDR stream pointed at by the **xdrs** parameter. The **addr** parameter points to the memory where the XDR stream's data is written to or read from. The **len** parameter specifies the length of the memory in bytes. The **x_op** parameter specifies the XDR direction. The possible choices are **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**.

1.2.332.1.9 Record Streams

Record streams are XDR streams built on top of record fragments, which are built on TCP/IP streams. TCP/IP is a connection protocol for a transporting large streams of data at one time, which contrasts with transporting a single data packet at a time. The primary use of a record stream is to interface remote procedure calls to TCP connections. It can also be used to stream data into or out of normal files.

XDR provides a record creation routine, **xdrrec_create**, for initializing record streams, and three routines for marking, or delimiting, the records in the data streams. These routines are discussed in the following:

```
#include <rpc/rpc.h>

void
xdrrec_create (xdrs, sendsize, recvsize, iohandle readproc, writeproc)
XDR *xdrs;
u_int sendsize;
u_int recvsize;
caddr_t iohandle;
int (*readproc) (), (*writeproc) ();
```

The **xdrrec_create** provides an XDR stream that can contain long sequences of records, and handle them in both the encoding and decoding directions. The record contents contain data in XDR form.

The **xdrs** parameter points to the XDR stream handle for the stream being called. The **sendsize** parameter sets the size of the input buffer where data is written to. The **recvsize** parameter sets the size of the output buffer where data is read from. If no value is specified, the buffers are set to the system defaults.

The **iohandle** parameter points to the input/output buffer's handle, which is opaque. The **readproc** parameter points to the routine to call when a buffer needs to be filled, and the **writeproc** parameter points to the routine to call when a buffer needs to be flushed. These routines are similar to the **read** and **write** system calls.

The **readproc** and **writeproc** routines take the following form:

```
int
process (iohandle, buf, nbytes)
     char *iohandle;
     char *buf;
     int nbytes;
```

The **process** parameter identifies which routine (**read** or **write**) to call. The **readproc** routine reads the number of bytes set by the **nbytes** parameter and places the bytes into the buffer pointed to by the **buf** parameter. The **writeproc** writes to the data stream the number of bytes specified by the **nbytes** parameter from the buffer specified by the **buf** parameter.

```
boot_t
xdrrec_endofrecord (xdrs, flushnow)
XDR *xdrs;
boot_t flushnow;
```

The **xdrrec_endofrecord** routine causes the current outgoing data to

be marked as a record. The **xdrs** parameter points to the XDR stream handle.

If the **flushnow** flag is used as a parameter with the value TRUE, the stream's **writeproc** routine is called to write out the completed record. Otherwise, **writeproc** is called when the output buffer is full.

```
bool_t
xdrrec_skiprecord (xdrs)
XDR *xdrs;
```

The **xdrrec_skiprecord** routine causes the position of an input stream to move past the current record boundary to the beginning of the next record in the stream.

```
bool_t
xdrrec_eof (xdrs)
XDR *xdrs;
```

The **xdrrec_eof** routine checks the buffer for an input stream. It returns the value 1 when there is no more input in the stream's buffer. If it returns the value 0, there is more input in the stream.

Note: Do not confuse this to mean that there is no more data in the underlying file descriptor.

1.2.332.1.10 Implementation of New XDR Streams

XDR streams can be created and implemented by programmers. Implementors must make an XDR structure routine that includes operation routines available to clients using a creation routine.

The abstract data types required for programmers to implement their own XDR streams are provided in the following:

```
enum xdr_op { XDR_ENCODE=0, XDR_DECODE=1, XDR_FREE=2 };

typedef struct {
     enum xdr_op x_op;
     struct xdr_ops {
        bool_t  (*x_getlong) ();
        boot_t  (*x_putlong) ();
        boot_t  (*x_getbytes) ();
        boot_t  (*x_putbytes) ();
        u_int   (*x_getpostn) ();
        boot_t  (*x_setpostn) ();
        caddr_t (*x_inline) ();
        VOID    (*x_destroy) ();

     } *x_ops;
     caddr_t x_public;
     caddr_t x_private;
     caddr_t x_base;
     int   x_handy;
} XDR;
```

The **x_op** parameter specifies the current operation being performed on the stream. This field is important to the XDR primitives but does not affect the implementation of the stream because the stream's implementation does not depend on the value.

The following set of parameters are pointers to XDR stream manipulation routines:

x_getlong Gets long integer values from the data stream.

x_putlong Puts long integer values into the data stream.

x_getbytes Gets bytes from the data stream.

x_putbytes Puts bytes into the data stream.

x_getpostn Returns stream offset.

x_setpostn Repositions offset.

x_inline Points to internal data buffer which can be used for any purpose.

x_destroy Frees private data structure.

Macros for accessing the **x_getpostn**, **x_setposn**, and **x_destroy** routines are defined earlier in this XDR section. See "Non-Filter Primitives" in topic 1.2.332.1.4.

The **x_inline** operation is defined as follows:

```
long *
x_inline (xdrs, len)
XDR *xdrs;
int len;
```

This routine returns a pointer to an internal piece of the buffer of a stream, which is pointed to by the **xdrs** parameter. The **len** parameter specifies the size, in bytes, of the buffer. If the routine cannot find a buffer segment of the requested size, it may return the value 0. The buffer can be used for any purpose, but the buffer is not data-portable.

The **x_getbytes** operation gets byte sequences from the underlying XDR stream, while the **x_putbytes** operation puts byte sequences into the stream. The routines appear as follows:

```
bool_t
operation (xdrs, buf, bytecount)
XDR *xdrs;
char *buf;
u_int bytecount;
```

The **xdrs** parameter parameter points to the XDR stream handle. The **buf** parameter specifies the buffer and the **bytecount** specifies the size of the byte sequence being put or obtained from the data stream.

The **x_getlong** operation gets long numbers from the underlying XDR stream, while the **x_putlong** operation puts long numbers into the stream. These routines translate the numbers between the machine representation and the XDR representation. The routines appear as follows:

```
bool_t
operation (xdrs, lp)
XDR *xdrs;
long *lp
```

The **xdrs** parameter points to the XDR stream handle. The **lp** parameter points to the address of the stream receiving and acquiring the numbers. Upon successful completion, this routine returns the value TRUE. Otherwise, it returns the value FALSE.

Note: Higher level XDR implementations assume that signed and unsigned long integers contain the same number of bits, and that nonnegative integers and unsigned integers have the same bit representations.

The **x_public**, **x_private**, **x_base** and **x_handy** fields are specific to a stream's implementation. The **x_public** parameter specifies user data that is private to the stream's implementation and is not used by the XDR primitive that calls it. The **x_private** parameter is a pointer to the implementation data. The **x_base** parameter contains the position information in the data stream that is private to the user implementation. The **x_handy** data can contain extra information as necessary.

1.2.332.1.11 Passing Linked Lists Using XDR

This section describes how to pass linked lists of arbitrary length using XDR. To help illustrate the functions of the XDR routine for encoding, decoding, or freeing linked lists, the following example creates a data structure and defines its associated XDR routine before creating the linked list. In the example, a data structure and its associated XDR routines are created to list a person's gross assets and liabilities. The data structure and its associated XDR routine can be coded as follows:

```
struct gnumbers {
     long g_assets;
     long g_liabilities;
};

bool_t
xdr_gnumbers (xdrs, gp)
     XDR *xdrs;
     struct gnumbers *gp;
{
     if (xdr_long (xdrs,      &(gp->g_assets)))
          return (xdr_long (xdrs, &(           gp->g_liabilities)));
     return(FALSE);
}
```

The **xdrs** parameter points to the XDR data stream handle, and the **gp** parameter points to the address of the structure that provides the data to or from the XDR stream.

To create a linked list of the assets and liabilities structure, the following structure could be constructed:

```
typedef struct gnode {
     struct gnumbers gn_numbers;
     struct gnnode *nxt;
};

typedef struct gnnode *gnumbers_list;
```

The head of the linked list defines the data object. The **nxt** parameter specifies whether the object has terminated. If the object continues, the **nxt** parameter points to the address of the structure where the object continues. However, when the object is serialized, the link addresses no longer contain useful information.

The XDR data description of this linked list can be described by the recursive type declaration of the **gnumbers_list** as follows:

```
struct gnumbers {
     unsigned g_assets;
     unsigned g_liabilities;
};

typedef union switch (boolean) {
     case TRUE: struct {
          struct gnumbers current_element;
          gnumbers_list rest_of_list;
     };
     case FALSE: struct ();
} gnumbers_list:
```

In this description, the Boolean indicates whether data follows it. The value of the Boolean is FALSE when no data follows it. The value of the Boolean is TRUE, when it is followed by the rest of the object (**gnumbers** structure and **gnumbers_list**). The data structure has no Boolean explicitly declared in it; however the **nxt** parameter implicitly carries the information. Also note that the XDR data description has no pointer explicitly declared in it.

To write a set of XDR routines to serialize or deserialize a linked list of entries, you can use the XDR description of the data that has no pointer. The set contains the mutually recursive routines **xdr_gnumbers_list**, **xdr_wrap_list**, and **xdr_gnnode**, as the illustrated in the following:

```
bool_t
xdr_gnnode (xdrs, gp)
     XDR *xdrs;
     struct gnnode *gp;
{
     return (xdr_gnumbers (xdrs, &(     gp->gn_numbers))  &&
          xdr_gnumbers_list (xdrs, &(          gp->nxt)));

}
bool_t
xdr_wrap_list (xdrs, glp)
     XDR *xdrs;
     gnumbers_list *glp;
{
     return (xdr_reference (xdrs,      glp,
          sizeof(struct gnnode), xdr_gnnode));
}

struct xdr_discrim choices [2] = {
     /*
      * called if another node needs serializing or deserializing
      */
     { TRUE, xdr_wrap_list },
     /*
      * called when no more nodes need serializing or deserializing
      */
     { FALSE, xdr_void }
}

bool_t
xdr_gnumbers_list (xdrs, glp)
     XDR *xdrs;
     gnumbers_list *glp;
{
     bool_t more_data;

     more_data = (*glp != (gnumbers_list)NULL);
     return (xdr_union (xdrs, &more_data,
     glp, choices, NULL));
}
```

The entry routine **xdr_gnumbers_list** translates between the Boolean value **more_data** and the list pointer values. When there is no more data, the **xdr_union** routine calls the **xdr_void** routine which terminates the recursion. Otherwise, **xdr_union** calls **xdr_wrap_list** to dereference the list pointers. The **xdr_gnnode** routine actually serializes (or

deserializes) the data of the linked list's current node. It then recursively calls **xdr_gnumbers_list** to process the remainder of the list.

These routines function correctly in all three directions (**XDR_ENCODE**, **XDR_DECODE**, and **XDR_FREE**) for linked lengths of any lengths (including zero). The Boolean **more_data** is always initialized, but when **XDR_DECODE** is called, **more_data** is overwritten by an externally generated value. The value of the **bool_t** is lost in the stack, but its value is reflected in the pointers to the list.

Because of the recursion involved with these routines, when you use them to serialize or deserialize a list, the stack grows in linear proportion to the number of nodes in the list. The routines can become difficult to code because of the number of primitives involved. The following routine shows how to collapse the recursive routines:

```
bool_t
xdr_gnumbers_list (xdrs, glp)
     XDR *xdrs;
     gnumbers_list *glp;
{
     bool_t more_data;
     bool_t freeing
     gnumbers_list *next;  /*the next value of glp*/

     freeing = (xdrs->x_op ==XDR_FREE);
     while (TRUE) {
       more_data = (*glp !=        (gnumbers_list)NULL);
       if (!xdr_bool (xdrs, &more_data))
             return (FALSE) ;
       if (!more_data)
             return (TRUE) ;   /* we are done */
       if ( freeing)
             next = &((*glp)->nxt);
       if (!xdr_reference (xdrs,        glp sizeof(struct gnnode),
             xdr_gnumbers))
             return (FALSE);
       glp = (freeing) ? next : &((*glp)->nxt);
     }
}
```

The **glp** parameter acts as the address of the pointer to the head of the remainder of the list to be serializes or deserialized. The **glp** parameter is set to the address of the current node's **nxt** field at the end of the **while** loop. The discriminated union is implemented inline. The **more_data** variable is used in the same manner as in the preceding routines, and its value is recomputed and reserialized (or redeserialized) at each repeat of the loop. This example inspects the direction of the operation **(xdrs->x_op)**.

Since ***glp** is a pointer to a node, the pointer is dereferenced by the **xdr_reference** routine. The **sizeof** parameter sets the size of a node (data values plus the **nxt** pointer), while **xdr_numbers** only serializes or deserializes the data values.

Related Information

In this book: "Remote Procedure Call (RPC)" in topic 1.2.231.

The chapter on RPC in *AIX Operating System Programming Tools and*

Interfaces.

1.2.333 Network Information Service Client Interface

Purpose
Provides system information for networked NIS host machines.

Library
Internet Library (**libc.a**)

Syntax
 # include <rpcsvc/ypclnt.h>

Description
The IBM AIX Network File System NIS client interface is composed of the library functions used by the network information service to look up and return system information in a network that has the network information service installed. The network information service (NIS) is a distributed network lookup service that provides system information, such as user passwords and host names, to other machines running NIS. Read-only NIS data bases can be maintained on multiple nodes in a network of machines. Other network machines function as NIS clients by requesting information from the NIS data bases. See the section on network information service in *Managing the AIX Operating System* for more detailed information.

The NIS client interface handles client requests through the Remote Procedure Call (RPC) facility. RPC is a paradigm for interprocess communication that allows one process to call another process to execute functions in a remote environment. The input and output to NIS remote procedure calls are described using the External Data Representation (XDR) specification for representing basic data types in a uniform format. The XDR specification establishes standard representations for data types so that they can be transferred among machines without being dependent on machine type or architecture. See "Remote Procedure Call (RPC)" in topic 1.2.231 for more information on RPC, and "XDR (External Data Representation)" in topic 1.2.332 for more information on XDR.

Subtopics
1.2.333.1 How NIS Works
1.2.333.2 NIS Maps
1.2.333.3 NIS Client Interface Routines

1.2.333.1 How NIS Works

When NIS is configured into a network, the person who manages the system selects one server to be the NIS master server for a set of networked machines, and builds the master NIS data base on this server. After the master NIS data base is built, the NIS lookup service daemon, **ypserv**, is started. The master NIS data base is propagated, or replicated, to a number of other servers in the network that are designated as NIS slave servers. The **ypserv** daemons are started on these servers.

When NIS clients request system information, such as a host name, that information may reside on a remote NIS server, which means they are making a remote procedure call. The remote procedure call is sent to a **ypserv** daemon on one of the NIS servers. The **ypserv** daemon looks up the information in its local NIS data base and returns it to the client. See *AIX Operating System Commands Reference* and *Managing the AIX Operating System* for more information on the **ypserv** daemon.

1.2.333.2 NIS Maps

Network system information is stored in NIS maps. Each map contains data sets stored as keys and their associated values. The keys identify the names of such system information as hosts, passwords, and user IDs. For example, the hosts map contains the names and Internet addresses for all machines in the network. The host names are the keys and the Internet addresses are the values. The NIS lookup service matches the keys in the map to the name of the item requested by the client (the input string to the item) in order to return the associated value data to the client.

Each NIS map has a name that programs must use to access the data in the map. In addition, programs requesting information must know the format of the data in the NIS map. The format of the data in default NIS maps matches the format of the ASCII files used as input for the maps.

NIS maps are grouped by NIS domain. A NIS domain refers collectively to a designated group of hosts. Maps are stored in a subdirectory of **/etc/yp** that corresponds to the appropriate domain. For example, the NIS maps for hosts in the **publications** domain are stored in **/etc/yp/publications** on the NIS servers. To find the correct map, the **ypserv** daemon uses the **publications** domain name when calling map lookup functions.

1.2.333.3 NIS Client Interface Routines

The NIS client interface is composed of the library functions used by the **ypserv** daemon to look up information and to return it to the client. NIS uses matching and enumeration routines to get information from its data bases and return it to the clients. The system calls and parameters that are used for these functions are discussed in the following sections.

It is important to note that NIS clients must be bound to a NIS server that services the appropriate domain in order to use NIS services. When a client requests information, the lookup processes call the **ypbind** daemon to bind the client to a NIS server that can service the request. It is also possible to directly bind a client with a **yp_bind** routine. This is useful for programs that use backup strategies for instances when NIS is not available. For that reason, the **yp_bind** routine (and the contrasting **yp_unbind** routine) are listed at the end of this section.

A Note about Parameters: Memory is allocated to the NIS client interface routines using the **malloc** system call and can be freed if the user code has no continuing need for it.

Parameters with the prefix **in** are input parameters. The values of the **indomain** and **inmap** parameters cannot be null but must be null-terminated.

Parameters with the prefix **out** are output parameters. Output parameters of the **char **** type are addresses of uninitialized character pointers. The **outkey** and **outval** parameters cause 2 extra bytes of memory to be allocated at the end that contain new-line escape characters or the NULL value. The 2 bytes are not reflected in the **outkeylen** and **outvallen** parameter values.

String parameters that are accompanied by a count parameter cannot be null. They can point to null strings if the count parameter specifies it. Counted strings do not have to be null-terminated.

Enumerating Map Values:

```
yp_first (indomain, inmap, outkey, outkeylen, outval, outvallen)
char *indomain;
char *inmap;
char **outkey;
int *outkeylen;
char **outval;
int *outvallen;

yp_next (indomain, inmap, inkey, inkeylen, outkey, outkeylen,
     outval, outvallen)
char *indomain;
char *inmap;
char *inkey
int inkeylen
char **outkey;
int *outkeylen;
char **outval;
int *outvallen;
```

The **yp_first** routine returns the first value from the map it associates with the named key.

The **yp_next** routine returns each subsequent value it finds in the

named map until it reaches the end of the list.

The **indomain** and **inmap** parameters point to the names of the domain and map used as input to the system call.

In the **yp_first** routine the **inkey** parameter is not needed since the value returned is considered the first key.

The **yp_next** routine must be preceded by an initial **yp_first** call. Use the **outkey** value returned from the initial **yp_first** as the value of the **inkey** parameter for **yp_next**. The **inkey** values for subsequent calls are retrieved as the nth + second key-value pair. That is, each time the routine returns a key-value pair, the **outkey** value returned becomes the value to use as the next **inkey** parameter.

The **inkeylen** parameter is the length, in bytes, of the string **inkey**.

The **outval** parameter is an uninitialized pointer to a buffer where the values associated with the key are placed. The **outvallen** parameter is the length, in bytes, of the string pointed to by **outval**.

The concepts of **first** and **next** depend on the structure of the NIS map being processed. The routines do not retrieve the information in a specific order, such as the lexical order from the original data base information files or the numerical sorting order of the keys, values, or key-value pairs. They do show every entry in the NIS map if the **yp_first** function is called on a specific map with **yp_next** called repeatedly. The process returns the message **NISERR_NOMORE** to the user to indicate that every entry in the NIS map has been seen once. If the same sequence of operations is performed on the same map at the same server, the entries are seen in the same order.

Note: If a server operates under a heavy load or fails, the domain can become unbound and then bound again while a client is running. If it binds itself to a different server, it can cause entries to be seen twice or not be seen at all. The domain rebinds itself to protect the enumeration process from being interrupted before it completes. You can avoid this situation by returning all of the keys and values with the following **yp_all** routine.

```
yp_all (indomain, inmap, incallback)
char *indomain;
char *inmap;
struct ypall_callback incallback {
        int (* foreach) ();
        char *data;
};

foreach (instatus, inkey, inkeylen, inval, invallen, indata)
int instatus;
char *inkey;
int inkeylen;
char *inval;
int invalllen;
char *indata;
```

The **yp_all** routine transfers all of the key-value pairs from the NIS server to the client as the entire map. It uses TCP (rather than UDP) to transport the data. The entire transaction takes places as a single remote procedure call and response function.

The **indomain** and **inmap** parameters point to the names of the domain and map used as input to the system call.

The **instatus** parameter of the **foreach** routine holds a return status value of the form **NIS_TRUE** or an error code. The error codes are defined in the **<rpcsvc/yp_prot.h>** header file.

The **inkey** parameter points to memory that is private to the **yp_all** function and is overwritten when each new key-value pair arrives. The **foreach** function can use the contents of the memory but does not own the memory itself. Key and value objects presented to **foreach** look exactly like they do in the server's map. Objects not terminated by NEWLINE or NULL in the server's map are not terminated by NEWLINE or NULL in the client's map.

The **indata** parameter holds the contents of the **incallback->data** element passed to the **yp_all** routine. The **data** element shares state information between the **foreach** function and the mainline code. It is an optional parameter because no part of the NIS client package inspects its contents.

Since the **foreach** function is a Boolean, it returns 0 (zero) to indicate that it wants to be called again for additional received key-value pairs. It returns a nonzero value to stop the flow of key-value pairs. If **foreach** returns a nonzero value, it is not called again, and **yp_all** returns a value of 0 (zero).

Working with Domains and Maps:

```
yp_get_default_domain (outdomain)
char **outdomain;
```

NIS lookup calls require a map name and a domain name. The client processes can get the default domain of the node by calling the **yp_get_default_domain** routine and using the value returned by the **outdomain** parameter as the input domain (**indomain**) parameter for its NIS remote procedure calls.

```
yp_master (indomain, inmap, outname)
char *indomain;
char *inmap;
char **outname;
```

The **yp_master** routine returns the machine name of the NIS master server for a map.

The **indomain** and **inmap** parameters point to the names of the domain and map used as input to the system call. The **outname** parameter points to the character string that identifies the NIS master server by name.

```
yp_match (indomain, inmap, inkey, inkeylen, outval, outvallen)
char *indomain;
char *inmap;
char *inkey;
```

```
int inkeylen;
char **outval;
int *outvallen;
```

The **yp_match** routine searches for the value associated with a key. The input character string entered as the key must match a key in the NIS map exactly because pattern matching is not available in the NIS.

The **indomain**, **inmap**, and **inkey** parameters point to the names of the domain, map, and key used as input to the function. The **inkeylen** parameter is the length, in bytes, of the string **inkey**.

The **outval** parameter is an uninitialized pointer to a buffer where the values associated with the key are placed. The **outvallen** parameter is the length, in bytes, of the string pointed to by **outval**.

```
yp_order (indomain, inmap, outorder)
char *indomain;
char *inmap;
unsigned long *outorder;
```

The **yp_order** routine returns the order number for a map which identifies when the map was built. This is important in determining if the local map is the most current version or if the master NIS data base has a more current one.

The **indomain** and **inmap** parameters point to the names of the domain and map used as input to the system call.

The **outorder** parameter points to the order number, which is a ten-digit ASCII integer that represents the AIX time, in seconds, when the map was built.

Error Information:

```
char *yperr_string (incode)
int incode;
```

The **yperr_string** routine returns a pointer to an error message string. The error message string is null-terminated but contains no period or new-line escape characters.

```
ypprot_err (incode)
u_int incode;
```

The **ypprot_err** routine takes a NIS protocol error code as input, and returns an error code to be used as input to an **yperr_string** call.

Binding and Unbinding NIS Service:

```
yp_bind (indomain)
char *indomain;

void yp_unbind (indomain)
char *indomain;
```

In order to use the NIS services, the client process must be bound to a NIS server that serves the appropriate domain. That is, the

client must be associated with a specific NIS server that services the client's requests for NIS information. The NIS lookup processes automatically use the **ypbind** daemon to bind the client, but **yp_bind** can be used in programs to call the daemon directly for processes that use backup strategies when NIS is not available.

Each NIS binding allocates, or uses up, one client process socket descriptor, and each bound domain uses one socket descriptor. Multiple requests to the same domain use the same descriptor.

The **yp_unbind** routine is available to manage socket descriptors for processes that access multiple domains. When **yp_unbind** is used to free a domain, all per-process and per-node resources that were used to bind it are also freed.

Note: If an RPC failure status returns from the use of **yp_bind**, the domain is unbound automatically. When this occurs, the NIS client tries to complete the operation if the **ypbind** daemon is running and one of the following is true:

The client process cannot bind a server for the proper domain. Remote procedure calls to the server fail.

The NIS client returns control to the user with either an error or success code with results if any of the following occurs:

The error is not related to RPC.
The **ypbind** daemon is not running.
The **ypserv** daemon returns an answer.

Return Value
Upon successful completion, the NIS client interface routines return the value 0. Otherwise, they return a failure code with the prefix **NISERR_**.

Error Conditions
The NIS function calls fail if one or more of the following is true:

NISERR_BADARGS	Arguments to the function are not correct.
NISERR_RPC	Unable to complete remote procedure call because domain is not bound.
NISERR_DOMAIN	Cannot bind to NIS server on this domain.
NISERR_MAP	Map is not in server's domain.
NISERR_KEY	Key is not in map.
NISERR_NISERR	Internal NIS server or client error.
NISERR_RESRC	Resource allocation failure.
NISERR_NOMORE	No more records in map data base.
NISERR_PMAP	Cannot communicate with **portmap** daemon.
NISERR_NISBIND	Cannot communicate with **ypbind** daemon.
NISERR_NISSERV	Cannot communicate with the **ypserv** daemon.

NISERR_NODOM Local NIS domain name not set.

Files

/usr/include/rpcsvc/ypclnt.h
/usr/include/rpcsvc/yp_prot.h

Related Information
In this book: "Remote Procedure Call (RPC)" in topic 1.2.231 and "XDR (External Data Representation)" in topic 1.2.332.

The **ypbind**, **ypserv**, and administrative commands in *AIX Operating System Commands Reference*.

2.0 Volume 2. Files and Device Drivers

Subtopics
2.3 Chapter 3. File Formats
2.4 Chapter 4. Miscellaneous Facilities
2.5 Chapter 5. Special Files
2.6 Chapter 6. Advanced Display Graphics Support Library

2.3 Chapter 3. File Formats

Subtopics

2.3.1 About This Chapter

This chapter outlines the formats of various files. The C language **struct** declarations for the file formats are given where applicable. These structures are usually found in header files located in the **/usr/include** or **/usr/include/sys** directories, although they can be located in any directory in the file system.

Many of the files described in this chapter contain magic numbers at predefined offsets. Magic numbers provide programs with a way to verify the format of an input file before attempting to process it. The values used for magic numbers are chosen because they are not likely to occur as a random pattern in normal input.

2.3.2 a.out

Purpose
Provides common assembler and link editor output.

Synopsis
#include <a.out.h>

Description
The **as** (assembler), compilers (C, VS Pascal, and VS FORTRAN), and **ld** (link editor) programs produce an output file (the **a.out** file by default). The **a.out** file is executable if the assembler, compilers, and the link editor do not find any unresolved external references or errors in the source file.

This file can consist of the following sections: a file header, an auxiliary header, section headers for each of the file's raw data sections, the raw data sections, relocation data for each raw data section, line number information for each raw data section, a symbol table section, and, if long symbols are used, a strings table. A diagram of this structure follows.

Every **a.out** does not contain all the sections enumerated. In particular, the line number, symbol table, and strings table sections are not present if the program is linked with the **-s** flag of the **ld** command or if they were removed by the **strip** command.

Comment sections are only present as a result of certain as directives. Library sections (**.lib**) are only found in shared library archive members or programs linked with shared libraries. Initialization sections (**.init**) are only found in shared library archive members or programs linked with shared libraries that are not yet executable. Finally, there are no relocation sections if the file is executable.

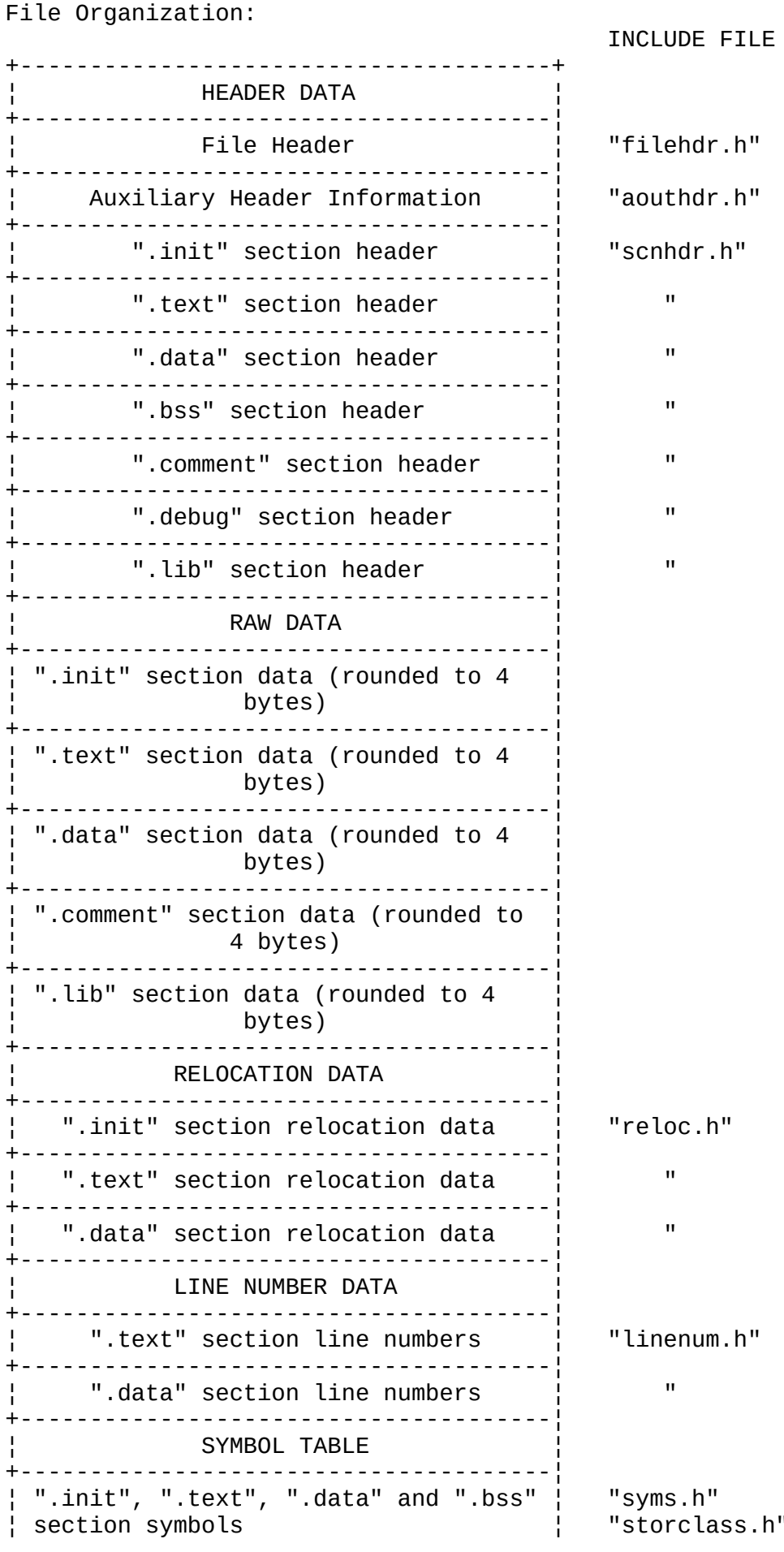
Loading an **a.out** file into memory for execution causes the creation of three logical segments: the text segment, the data segment (initialized data followed by data that is not initialized, the latter effectively being initialized to all zeros), and a stack.

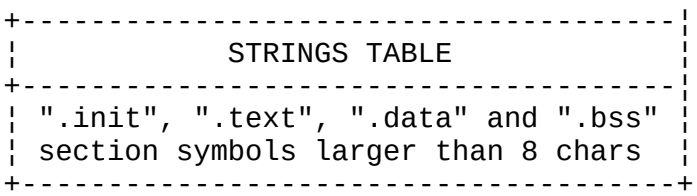
The text segment occupies a low memory address in the process image, and its size is static. The data segment follows the text segment. The size of this segment can be extended using the **brk** system call. The stack segment begins near the highest locations and grows toward the data segment as required.

Subtopics

2.3.2.1 Common Object File Format

File Organization:





Object File Components:

Header Files:

/usr/include/filehdr.h
/usr/include/aouthdr.h
/usr/include/scnhdr.h
/usr/include/reloc.h
/usr/include/linenum.h
/usr/include/syms.h
/usr/include/storclass.h

Standard File (includes the header files above):

/usr/include/a.out.h Object file

2.3.2.2 File Header

The format of the file header is as follows:

```
struct filehdr {
  unsigned short    f_magic;          /* magic number */
  unsigned short    f_nscns;          /* number of sections */
  long              f_timdat;         /* time & date stamp */
  long              f_symptr;         /* file pointer to symtab */
  long              f_nsyms;          /* number of symtab entries */
  unsigned short    f_opthdr;         /* sizeof(optional hdr) */
  unsigned short    f_flags;          /* flags */
  };
```

The fields in the file header are defined as follows:

f_magic A 2-byte machine type identification number.

f_nscns The number of sections in this file.

f_timdat A 4-byte number encoding the time and date that the file was created.

f_symptr A file pointer or offset into the file to the start of the symbol table section.

f_nsyms The number of symbol table entries.

f_opthdr The size of the auxiliary or optional header. This field is 0 if there is no header.

f_flags The bits in the **f_flags** field are defined as follows:

F_RELFLG Relocation information stripped from file

F_EXEC File is executable (that is, no unresolved external references)

F_LNNO Line numbers stripped from file

F_LSYMS Local symbols stripped from file

F_MINIMAL This is a minimal object file (***.m**) output of **fextract**.

F_SWABD This file has had its bytes swabbed (in names).

F_UPDATE This is a fully bound update file output of **ogen**.

F_AR16WR This file created on AR16WR machine (for instance, Intel 80286)

F_AR32WR This file created on AR32WR machine (for instance, IBM PS/2)

F_AR32W This file created on AR32W machine (for instance, IBM System/370)

F_PATCH File contains "patch" list in optional header

F_NODF (Minimal file only) no decision functions for replaced functions.

The numerical value of the flags is as follows:

```
#define         F_RELFLG        0000001
#define         F_EXEC          0000002
#define         F_LNNO          0000004
#define         F_LSYMS         0000010
#define         F_MINMAL        0000020
#define         F_UPDATE        0000040
#define         F_SWABD         0000100
#define         F_AR16WR        0000200
#define         F_AR32WR        0000400
#define         F_AR32W         0001000
#define         F_PATCH         0002000
#define         F_NODF          0002000
```

2.3.2.3 Auxiliary Header

The format of the auxiliary header is as follows:

```
typedef struct aouthdr {
  short  magic;      /* magic number - see /etc/magic */
  short  vstamp;     /* version stamp   */
  long   tsize;      /* text size in bytes, padded to FW boundary */
  long   dsize;      /* initialized data "  "  */
  long   bsize;      /* uninitialized data "   "  */
  long   entry;      /* entry pt.    */
  long   text_start; /* base of text used for this file */
  long   data_start; /* base of data used for this file */
} AOUTHDR;
```

The fields in the auxiliary header are defined as follows:

magic File type identification number.

vstamp Number used to identify the file version. The linker fills in this field upon request.

tsize The size of the **.text** segment, rounded to the nearest 4-byte boundary.

dsize The size of the **.data** segment, rounded to the next 4-byte boundary.

bsize The size of the **.bss** segment, rounded to the next 4-byte boundary.

entry The starting address of the program at execution time.

text_start The base address of the **.text** segment at execution time

data_start The base address of the **.data** segment at execution time

2.3.2.4 Section Headers

Each raw data section of the **COFF** file has a corresponding section header with the following format:

```
    struct scnhdr {
      char  s_name[8];              /* section name */
      long  s_paddr;                /* physical address */
      long  s_vaddr;                /* virtual address */
      long  s_size;                 /* section size - in bytes */
      long  s_scnptr;               /* file ptr to raw data for section */
      long  s_relptr;               /* file ptr to relocation entries */
      long  s_lnnoptr;              /* file ptr to line numbers entries */
      unsigned short s_nreloc;      /* number of relocation entries */
      unsigned short s_nlnno;       /* number of line number entries */
      long  s_flags;                /* flags */
    };

/*
 * The low two bytes of s_flags is used as a section "type"
 */

  #define STYP_REG      0x00        /* "regular" section: */
                                    /*    allocated, relocated, loaded */
  #define STYP_DSECT    0x01        /* "dummy" section: */
                                    /*    not allocated, relocated,
                                          not loaded */
  #define STYP_NOLOAD   0x02        /* "noload" section: */
                                    /*    allocated, relocated,
                                          not loaded */
  #define STYP_GROUP    0x04        /* "grouped" section: */
                                    /*    formed of input sections */
  #define STYP_PAD      0x08        /* "padding" section: */
                                    /*    not allocated, not relocated,
                                          loaded */
  #define STYP_COPY     0x10        /* "copy" section: */
                                    /*for decision function used by
                                          field update;
                                    /*    not allocated, not relocated,
                                          loaded;  reloc & lineno
                                          entries processed normally */
  #define STYP_TEXT     0x20        /* section contains text only */
  #define STYP_DATA     0x40        /* section contains data only */
  #define STYP_BSS      0x80        /* section contains bss only */
  #define S_NEWFUN      0X100       /* new function in an update file*/
  #define STYP_INFO     0x200       /* comment section : not allocated
                                          not relocated, not loaded */
  #define STYP_OVER     0x400       /* overlay section : relocated
                                          not allocated or loaded */
  #define STYP_LIB      0x800       /* for .lib section : same as INFO */
```

Note: The physical address of a section is its offset from address zero of the address space. It is NOT necessarily the run-time address.

2.3.2.5 Relocation Data

A word in the text or data segment of memory contains either an actual value or the value of an offset. If a word in the text or data segment references an undefined external symbol, its value is an offset from the associated external symbol. During processing, the link editor defines the external symbol and adds the value of the symbol to the word in the file.

When relocation information is present, each item that can be relocated is 10 bytes long. The format of the relocation information is as follows:

```
struct reloc {
   long r_vaddr;           /* (virtual) address of reference */
   long r_symndx;          /* index into symbol table */
   unsigned short r_type;  /* relocation type */
};
```

The **r_vaddr** field gives the location of the relocatable reference relative to the beginning of the segment in which it is defined.

The **r_symndex** field contains an index, counted from zero, of the symbol table entry that is referenced.

The **r_type** field indicates to the link editor the type of relocation that is to be performed on the relocation entry during the linking process.

The currently defined relocation types are as follows:

```
#define  R_ABS     0
#define  R_OFF8    07
#define  R_OFF16   010
#define  R_SEG12   011
#define  R_AUX     013
#define  R_DIR16   01
#define  R_REL16   02
#define  R_IND16   03
#define  R_DIR24   04
#define  R_REL24   05
#define  R_OPT16   014
#define  R_IND24   015
#define  R_IND32   016

#define  R_DIR10   025
#define  R_REL10   026
#define  R_REL32   027
#define  R_DIR32   06
#define  R_DIR32S  012

#define  R_RELBYTE 017
#define  R_RELWORD 020
#define  R_RELLONG 021
#define  R_PCRBYTE 022
#define  R_PCRWORD 023
#define  R_PCRLONG 024
```

2.3.2.6 Line Number Data

When present, there is one line number entry for every "breakpointable" source line in a section. Line numbers are grouped on a per function basis; the first entry in a function grouping will have **l_lnno = 0** and, in place of the physical address, there will be the symbol table index of the function name.

The format of a line number entry is as follows:

```
struct lineno
{
    union
    {
       long   l_symndx ; /* sym. table index of function name
                                        if l_lnno == 0      */
       long   l_paddr ;  /* (physical) address of line number */
    } l_addr ;
    unsigned short  l_lnno ;   /* line number */
    } ;
```

Line number entries are used by a symbolic debugger to debug code at the source level.

2.3.2.7 Symbol Table Data

The symbol table consists of the following entries:

```
    struct syment
    {
        union
        {
            char        _n_name[SYMNMLEN]; /* old COFF version */
            struct
            {
                long   _n_zeros;           /* new == 0 */
                long   _n_offset;          /* offset into string table */
            } _n_n;
            char        *_n_nptr[2];       /* allows for overlaying */
        } _n;
        unsigned long   n_value;           /* value of symbol */
        short           n_scnum;           /* section number */
        unsigned short  n_type;            /* type and derived type */
        char            n_sclass;          /* storage class */
        char            n_numaux;          /* number of aux. entries */
    };
        #define SYMNMLEN   8    /* Number of characters in a symbol name */
        #define SYMENT struct syment
        #define SYMESZ    18    /* sizeof(SYMENT) - on disk */
                                /* WARNING: size may differ in memory */
                                /* because of alignment restrictions */

        #define n_name          _n._n_name
        #define n_nptr          _n._n_nptr[1]
        #define n_zeros        _n._n_n._n_zeros
        #define n_offset        _n._n_n._n_offset
```

The fields in the symbol table are defined as follows:

_n_name:
The symbol name, which consists of a null-terminated ASCII string, is contained within the **_n_name** field,if the length of the name is less than equal to **SYMNMLEN**. Otherwise, the first 4 bytes of the field are zero and the next 4 bytes contain a file pointer into the strings table section, which contains the name.

2.3.2.8 Symbol Value

n_value:

The value assigned to a symbol is dependent upon its storage class. (See "Storage Classes" in topic 2.3.2.9.) In the usual case of interest, relocatable symbols have a value that is equal to the virtual address of the symbol. Of course, when a program module is linked with other modules this value changes.

The full meaning of **n_value** is summarized in the following table:

Storage Class	**n_value**
C_AUTO	stack offset in bytes
C_EXT	relocatable address
C_STAT	relocatable address
C_REG	register number
C_LABEL	relocatable address
C_MOS	offset in bytes
C_ARG	stack offset in bytes
C_STRTAG	0
C_MOU	0
C_UNTAG	0
C_TPDEF	0
C_ENTAG	0
C_MOE	enumeration value
C_REGPARM	register number
C_FIELD	bit displacement
C_BLOCK	relocatable address
C_FCN	relocatable address
C_EOS	size
C_FILE	pointer to next **C_FILE** entry
C_ALIAS	tag index
C_HIDDEN	relocatable address

Note: The **C_FILE** entries form a singly linked list in the symbol table. For the last entry in the symbol table, the value of the symbol is the index of the first global symbol.

n_scnum:

Relocatable symbols have a section number, **n_scnum**, of the section in which they are defined. Otherwise, section numbers have the following meanings:

```
#define  N_UNDEF            0   /* undefined symbol */
#define  N_ABS             -1   /* value of symbol is absolute */
#define  N_DEBUG           -2   /* special debugging symbol
                                   value of symbol is meaningless
#define  N_TV
         (unsigned short)  -3   /* symbol needs transfer vector
                                   (preload) */
#define  P_TV
         (unsigned short)  -4   /* symbol needs
                                   transfer vector (postload) */
```

n_type:

The **n_type** field is primarily for use by a symbolic debugger.

The fundamental type of a symbol packed into the low 4 bits of the **n_type** field as follows :

```
#define  T_NULL     0
#define  T_ARG      1          /* function argument
                                  (only used by compiler) */
#define  T_CHAR     2          /* character */
#define  T_SHORT    3          /* short integer */
#define  T_INT      4          /* integer */
#define  T_LONG     5          /* long integer */
#define  T_FLOAT    6          /* floating point */
#define  T_DOUBLE   7          /* double word */
#define  T_STRUCT   8          /* structure  */
#define  T_UNION    9          /* union  */
#define  T_ENUM     10         /* enumeration  */
#define  T_MOE      11         /* member of enumeration */
#define  T_UCHAR    12         /* unsigned character */
#define  T_USHORT   13         /* unsigned short */
#define  T_UINT     14         /* unsigned integer */
#define  T_ULONG    15         /* unsigned long */
```

The high-order bits form the derived type. The derived types are defined as follows:

```
#define  DT_NON      0          /* no derived type */
#define  DT_PTR      1          /* pointer */
#define  DT_FCN      2          /* function */
#define  DT_ARY      3          /* array */
```

Type packing constants are defined as follows:

```
#define  N_BTMASK     017
#define  N_TMASK      060
#define  N_TMASK1     0300
#define  N_TMASK2     0360
#define  N_BTSHFT     4
#define  N_TSHIFT     2
```

2.3.2.9 Storage Classes

n_sclass:

The field **n_sclass** has one of the following values:

```
#define  C_EFCN          -1    /* physical end of function */
#define  C_NULL          0
#define  C_AUTO          1     /* automatic variable */
#define  C_EXT           2     /* external symbol */
#define  C_STAT          3     /* static */
#define  C_REG           4     /* register variable */
#define  C_EXTDEF        5     /* external definition */
#define  C_LABEL         6     /* label */
#define  C_ULABEL        7     /* undefined label */
#define  C_MOS           8     /* member of structure */
#define  C_ARG           9     /* function argument */
#define  C_STRTAG        10    /* structure tag */
#define  C_MOU           11    /* member of union */
#define  C_UNTAG         12    /* union tag */
#define  C_TPDEF         13    /* type definition */
#define  C_USTATIC       14    /* undefined static */
#define  C_ENTAG         15    /* enumeration tag */
#define  C_MOE           16    /* member of enumeration */
#define  C_REGPARM       17    /* register parameter */
#define  C_FIELD         18    /* bit field */
#define  C_WKEXT         20    /* Fortran weak extern */
#define  C_BLOCK         100   /* ".bb" or ".eb" */
#define  C_FCN           101   /* ".bf" or ".ef" */
#define  C_EOS           102   /* end of structure */
#define  C_FILE          103   /* file name */

#define  C_ALIAS         105   /* duplicate tag */
#define  C_HIDDEN        106   /* special storage class for extern
                                  symbols in dmert public librarie
#define  C_ENDINIT       107   /* special storage class for
                                  .bei and .fei sdb information
#define  C_LFNE          104   /* dummy for line number entries
                                  reformatted as symbol table entr
```

2.3.2.10 Auxiliary Entry Format

n_numaux:

The **n_numaux** field contains the number of auxiliary entries associated with this symbol table entry. Currently, a symbol table entry can have at most one auxiliary entry.

The auxiliary entry provides additional information, and has this format:

```
union auxent
{
    struct
    {
        long       x_tagndx; /* str, union, or enum tag indx */
        union
        {
            struct
            {
                unsigned short  x_lnno; /* declaration line number
                unsigned short  x_size; /* str, union, array size
            } x_lnsz;
            long   x_fsize;  /* size of function */
        } x_misc;
        union
        {
            struct   /* if ISFCN, tag, or .bb */
            {
                long    x_lnnoptr; /* ptr to fcn line # */
                long    x_endndx;  /* entry ndx past block end */
            } x_fcn;
            struct          /* if ISARY, up to 4 dimen. */
            {
                unsigned short    x_dimen[DIMNUM];
            } x_ary;
        } x_fcnary;
        unsigned short  x_tvndx;  /* tv index */
    } x_sym;
    struct
    {
        char    x_fname[FILNMLEN];
    } x_file;
    struct


    {
        long    x_scnlen;          /* section length */
        unsigned short  x_nreloc;  /* number of relocation entries
        unsigned short  x_nlinno;  /* number of line numbers */
    } x_scn;

    struct
    {
        long          x_tvfill;          /* tv fill value */
        unsigned short    x_tvlen;       /* length of .tv */
        unsigned short    x_tvran[2];     /* tv range */
    } x_tv;  /* info about .tv section (in auxent of symbol .tv))
};
```

```
#define AUXENT union auxent
#define AUXESZ 18    /* sizeof(AUXENT) */

#define FILNMLEN 14  /* Number of characters in a file name */
#define DIMNUM  4    /* Number of array dimensions in aux entry */

/* Defines for "special" symbols   */

#define _ETEXT    "etext"
#define _EDATA    "edata"
#define _END      "end"
```

The information in an auxiliary entry cannot be correctly interpreted without the symbol table entry to which it belongs. The order of entries within the symbol table is significant.

2.3.2.11 Strings Table

The strings table contains the names of symbols that are longer than 8 characters. If present, the first 4 bytes contain the length, in bytes, of the strings table, including the length bytes. Thus, offsets into the strings table are greater than or equal to 4. The remainder of the table is a sequence of null-terminated ASCII strings. If the **n_zeros** field in a symbol table entry is 0, the **n_offset** field gives the offset into the strings table of the name for the symbol.

2.3.2.12 Access Routines

To ease program access to the **COFF** file sections, you should use the access routines in the library **libld**. Within the library **libld**, there exist routines to open, close, and access sections of a **COFF** file.

Related Information
In Volume 1 of this book see: "ldahread" in topic 1.2.141, "ldclose, ldaclose" in topic 1.2.142, "ldfcn" in topic 1.2.143, "ldfhread" in topic 1.2.144, "ldgetname" in topic 1.2.145, "ldlread, ldlinit, ldlitem" in topic 1.2.146, "ldlseek, ldnlseek" in topic 1.2.147, "ldohseek" in topic 1.2.148, "ldopen, ldaopen" in topic 1.2.149, "ldrseek, ldnrseek" in topic 1.2.150, "ldshread, ldnshread" in topic 1.2.151, "ldsseek, ldnsseek" in topic 1.2.152, "ldtbindex" in topic 1.2.153, "ldtbread" in topic 1.2.154, and "ldtbseek" in topic 1.2.155.

The **config** program and the **as**, **cc**, **dump**, **ld**, **nm**, **dbx**, **size**, **strip**, and **what** commands in the *AIX Operating System Commands Reference*.

2.3.3 acct

Purpose
Provides the accounting file format for each process.

Synopsis
#include <sys/acct.h>

Description
The accounting files provide a means to monitor the use of the system. These files also serve as a method for billing each process for processor usage, materials, and services. The **acct** system call produces accounting files. The **<sys/acct.h>** file defines the records in these files. The content of the records are:

```
/* Accounting structures */
typedef ushort comp_t;     /* floating point */
          /* 13-bit fraction, 3-bit exponent */

struct acct
{
     char  ac_flag;     /* Accounting flag */
     char  ac_stat;     /* Exit status */
     ushort  ac_uid;    /* Accounting user-ID */
     ushort  ac_gid;    /* Accounting group-ID */
     dev_t  ac_tty;     /* control typewriter */
     time_t  ac_btime;  /* Beginning time */
     comp_t  ac_utime;  /* accounting user time in clock ticks */
     comp_t  ac_stime;  /* accounting system time in clock ticks */
     comp_t  ac_etime;  /* accounting elapsed time in clock ticks */
     comp_t  ac_mem;    /* memory usage */
     comp_t  ac_io;     /* chars transferred */
     comp_t  ac_rw;     /* blocks read or written */
     char  ac_comm[8];  /* command name */
};

#define AFORK  01           /* has executed fork, but no exec */
#define ASU    02           /* used superuser authority */
#define ACCTF  0300         /* record type: 00 = acct */
```

The fields are as follows:

ac_comm This field contains the command name. A child process, created by a **fork** system call, receives this information from the parent process. An **exec** system call resets this field.

ac_flag This field indicates whether the process used superuser authority, or whether it was created using a **fork** command but not yet followed by an **exec** system call. The **fork** command turns on the **AFORK** flag in this field and the **exec** system call turns off the **AFORK** flag.

ac_mem This field contains memory usage. For each clock tick, the system updates this field with the current process size and charges usage time to the process. This is computed as (**(data size)** + **(text size)**) ÷ (**number of in-memory processes using text**)

The following structure (not part of **acct.h**) represents the total accounting format used by the various accounting commands:

```
/* Float arrays below contain prime time and non-prime time
   components */

   struct tacct {
     uid_t  ta_uid;            /* user-ID */
     char   ta_name[8];        /* login name */
     float  ta_cpu[2];         /* cum. CPU time, p/np (mins) */
     float  ta_kcore[2];       /* cum. kcore-mins, p/np */
     float  ta_io[2];          /* cum. chars xferred (512s) */
     float  ta_rw[2];          /* cum. blocks read/written */
     float  ta_con[2];         /* cum. connect time, p/np, mins */
     float  ta_du;             /* cum. disk usage */
     long   ta_qsys;           /* queuing sys charges (pgs) */
     float  ta_fee;            /* fee for special services */
     long   ta_pc;             /* count of processes */
     unsigned short  ta_sc;    /* count of login sessions */
     unsigned short  ta_dc;    /* count of disk samples */
   };
```

File
/usr/include/sys/acct.h

Related Information
In this book: "acct" in topic 1.2.11 and "utmp, wtmp, .ilog" in topic 2.3.60.

The **acctcom** command in *AIX Operating System Commands Reference*.

The **acct**, **acctcms**, **acctcon**, **acctmerg**, **acctprc**, **diskusg,** and **runacct** procedures in *AIX Operating System Commands Reference*.

2.3.4 ar

Purpose
Describes common archive file format.

Synopsis
#include <ar.h>

Description
The **ar** (archive) command is used to combine several files into one. The **ar** command creates an **ar** file. The **ld** (link editor) searches archive files to resolve program linkage.

Each archive begins with the archive magic string:

```
#define ARMAG  "!<arch>\n"   /* magic string */
#define SARMAG  8            /* length of magic string */
```

Each archive that contains common object files includes an archive symbol table. See "a.out" in topic 2.3.2 for the format of an object file. **ld** uses this symbol table to determine the archive members to load during the link edit process. The archive symbol table, if it exists, is always the first file in the archive. It is never listed, but **ar** automatically creates and updates it.

The archive file members follow the archive header and symbol table. A file member follows each file member header. The format of a file member header is:

```
#define ARFMAG    "\n"      /* header trailer string */

struct ar_hdr {       /* file member header */
  char  ar_name[16];  /* file member name - terminated by '/'*/
  char  ar_date[12];  /* file member date */
  char  ar_uid[6];    /* file member user identification */
  char  ar_gid[6];    /* file member group identification */
  char  ar_mode[8];   /* file member mode */
  char  ar_size[10];  /* file member size */
  char  ar_fmag[2];   /* ARFMAG - string to end header */
};
```

All information in the file member header is in printable ASCII. The numeric information contained in the headers is stored as decimal numbers, except **ar_mode**, which is stored in octal. Thus, if the archive contains printable files, you can print the archive.

The **ar_name** field is blank-padded and terminated by a / (slash). The **ar_date** field indicates the date the file was last modified prior to archive. The **ar** command allows archives to move from system to system.

Each archive file member begins on an even-byte boundary. **ar** inserts null bytes for padding and a new-line character between files, if necessary. The **ar_size** field is the size of the file without padding. An archive file contains no empty areas.

If the archive symbol table exists, the first file in the archive has a zero-length name, for example, **ar_name[0] == '/'**. The contents of the symbol table are as follows:

The number of symbols. This is 4 bytes long.

The array of offsets into the archive file. The length is determined by 4 bytes times the number of symbols.

The name string table. The size is determined by **ar_size** minus (4 bytes times (the number of symbols plus 1)).

The **sgetl** and **sputl** functions manage the number of symbols and the array of offsets. The string table contains an equal number of null-terminated strings and elements in the offsets array. Each offset from the array associates with the corresponding name from the string table, in order. The string table names all the defined global symbols found in the object files contained in the archive. Each offset locates the archive header for the associated symbol.

Related Information
In this book: "sputl, sgetl" in topic 1.2.280 and "a.out" in topic 2.3.2.

The **ar**, **ld**, and **strip** commands in *AIX Operating System Commands Reference*.

2.3.5 attributes

Purpose
Describes an attribute file format.

Description
ASCII files are used to control some AIX utilities in order to simplify installing, customizing, and maintaining the PS/2 and System/370. A text editor can be used to examine or change these files. These files share an attribute-structured format.

An attribute-structured file consists of one or more named stanzas, separated by blank lines. Each stanza begins with a name followed by a colon, and contains assignments to keyword **attributes**. The values assigned can be alphanumeric strings or arbitrary character strings enclosed in double quotes. The assignments can associate either a single value or a succession of values with each attribute.

Note: In the multibyte environment, all attribute-structured files distributed with your system contain only ASCII characters. However, the library routines can handle multibyte characters, and provided that all parties who use the stanza values are operating in the same locale, multibyte characters can be used in these file stanzas.

The size limits for stanzas are: a maximum of 400 keywords per stanza, a maximum of 4K bytes per stanza, and a maximum of 512 bytes per keyword.

Typically, the stanza name is the name of a device or service. The attributes describe the properties or handling of the named device or service. The meanings of the stanza names, **attribute** names, and values are specific to an application. Examples of this file type distributed with the system are **/etc/filesystems**, **/etc/ports**, and **/etc/qconfig**.

The stanza name **default** can be used to specify default values for any **attributes**. These default assignments are implicitly included in every stanza that follows. A specified value overrides the default value. A new **default** stanza automatically cancels all previously specified default values. The syntax of a file of this type is:

```
<file>          ::=  <stanza>
                ::=  <file> <blank line> <stanza>
<stanza>        ::=  <name>:
                ::=  <name>:<assignments>
<name>          ::=  file name or information similar in syntax
<assignments>   ::=  <assignment>
                ::=  <assignments> <assignment>
<assignment>    ::=  <attribute>=<values>
<attribute>     ::=  string of alphanumeric characters
<values>        ::=  <value>
                ::=  <values>,<value>
<value>         ::=  string of alphanumeric characters
                ::=  "arbitrary characters"
```

Lines beginning with an * (asterisk) are considered to be comments and are ignored.

Note: Make sure that the values assigned to attribute keywords do not contain blanks unless they are enclosed in double quotes.

Related Information

In this book: "cc.cfg" in topic 2.3.8, "connect.con" in topic 2.3.9, "filesystems" in topic 2.3.18, "master" in topic 2.3.32, "ports" in topic 2.3.46, "qconfig" in topic 2.3.49, "rasconf" in topic 2.3.50, and "system" in topic 2.3.56.

AIX Guide to Multibyte Character Set (MBCS) Support.

2.3.6 autolog

Purpose
Performs the login function automatically.

Description
The optional **autolog** file causes the AIX system to perform a login sequence automatically when it contains a valid user name. When power is applied to the system and the login port is the console, **login** searches for this file. If this file is found, **login** creates a session for a specific user automatically. The **autolog** file is an ASCII file containing a valid user name. A system administrator can create this file while customizing the system. After the file is created, it can be edited with any editor. If this file does not exist, **login** causes the user to log in as usual.

If the Transparent Computing Facility is installed, this file should not be made common to all sites in the cluster. The way to accomplish this is to make **/etc/autolog** a symbolic link to **<LOCAL>/autolog**, and then create **autolog** only on the site or sites where an automatic login should occur. Care should be taken to ensure that each **autolog** file is writable only by system administrators.

File
/etc/autolog

Related Information
The **login** command in *AIX Operating System Commands Reference*.

2.3.7 backup

Purpose
Copies file system onto temporary storage media.

Synopsis
#include <backup.h>

Description
A backup of the file system provides protection against substantial data loss due to accidents or error. The **backup** command writes file system backups and, conversely, the **restore** command reads file system backups. The following text describes the format of a file system backup.

Subtopics

2.3.7.1 Header Types

The backup contains several different types of header records along with the data in each file that is backed up. The types of header records are:

FS_VOLUME The volume label. This header exists on every volume.

FS_DFINDEX A version of **FS_FINDEX** for compatibility with AIX/RT; created using the **-c** flag of the backup command.

FS_CLRI A bit map of inodes on the file system. A 0 bit indicates the inode is not in use. This header exists only on the first volume. If the backup is a level-zero backup, this header is omitted.

FS_BITS Another bit map of inodes. A one bit indicates the inode is present on this volume or a subsequent volume. This header may not appear on all volumes.

FS_VOLEND Indicates the end of the current volume. This header may not appear on all volumes. This header is used to indicate that all index entries on this volume are used.

FS_END Indicates the end of the backup. This header appears on every volume.

FS_DINODE A version of **FS_INODE** for compatibility with AIX/RT; created using the **-c** flag of the backup command.

FS_DNAME A version of **FS_NAME** for compatibility with AIX/RT; created using the **-c** flag of the backup command.

FS_DS Specifies the node ID of the node being backed up and the full path name of the directory that is being backed up. This header appears only on the first volume.

FS_INODE Describes a single inode. This header is followed by data that consists of directories then followed by the other files within the directories.

FS_NAME A description of a file that is backed up by name.

FS_FINDEX An index of files on this volume. Multiple headers of this type can appear on a volume if there are too many inodes for the initial index. This header is followed by data.

FS_BIGCLRI Similar to FS_CLRI, except used in cases where the number of inodes exceed 65,536.

FS_BIGBITS Similar to FS_BITS, except used in cases where the number of inodes exceed 65,536.

2.3.7.2 Header Sequence

The header sequence varies depending on whether the files are backed up by inode or by name and on the type of backup device used.

Volume 1 of inode backups to direct access volumes have the following sequence, assuming that more than one volume is required for backup:

FS_VOLUME
FS_CLRI
FS_BITS
FS_FINDEX, followed by data
FS_FINDEX (if applicable), followed by data
FS_END

Subsequent volumes have the following sequence:

FS_VOLUME, followed by data
FS_FINDEX, followed by data
FS_FINDEX (if applicable), followed by data
FS_END

Inode backups to tapes have the same format as previously described, except there are no **FS_FINDEX** headers and the **FS_BITS** header appears on every volume.

The format of backups by name does not depend on the output device. These backups have a simple format:

FS_VOLUME Appears on each volume.

FS_DS Appears on the first volume.

FS_NAME Precedes the data for each file. The files are copied in the order they were named.

FS_END Concludes the backup.

2.3.7.3 Header Format

The location and size of the headers are independent of any blocking for either the file system or the backup device. Each header begins on an 8-byte boundary. The length of a header depends on its type, but is always padded to a multiple of 8 bytes. Data from a file is similarly padded. Some headers contain addresses of other headers that are the offset in 8-byte units from the beginning of the backup volume.

Each field in a header is written in low-order bytes first for portability. Inode numbers within directories also follow this order. The header begins with the following structure:

```
struct hdr {
       unsigned char  len;
       unsigned char  type;
       ushort         magic;
       ushort         checksum;
};
```

The fields in this header indicate the following information:

len The length of the header in 8-byte units.

type The type of the header.

magic The magic number, which identifies this file as a file system backup. The magic number is one of the following values:

- **MAGIC** Identifies this as a regular file system backup.
- **PACKED_MAGIC** Identifies this as a packed, or compressed, file system backup. Each data file within it is compressed using the same algorithm that is used by the **pack** command. Header information is not compressed.

checksum A checksum.

2.3.7.4 Volume Headers

FS_VOLUME headers have the following structure:

```
struct {
      struct hdr h;
      ushort     volnum;
      time_t     date;
      time_t     budate;
      daddr_t    numwds;
      char       disk[16];
      char       fsname[16];
      char       user[16];
      short      incno;
 };
```

The fields contain the following information:

volnum Contains the volume number.

date Indicates the date the backup was made.

budate Indicates that all files changed since this date are backed up.

numwds Indicates the number of 8-byte words in this backup.

disk Identifies the device that was backed up.

fsname Identifies the logical name of the backed-up device, for example, **/a**.

user Identifies the user that made the backup.

incno Shows the level number of the backup.

For backups by name, **budate**, **disk**, and **fsname** have no meaning, and **incno** is 100.

2.3.7.5 Index Headers

FS_FINDEX and **FS_DFINDEX** records are as follows:

```
struct {
    struct hdr h;
    ushort     dummy;
    ino_t      ino[80];
    daddr_t    addr[80];
    daddr_t    link;
 };
```

The fields are:

ino I-numbers of files indexed

addr Addresses of file indexed

link Address of next index on this volume, or 0 if this is the last.

2.3.7.6 Bit Maps

FS_CLRI and **FS_BITS** headers have the same structure:

```
struct {
   struct hdr h;
   ushort     nwds;
};
```

In both cases, the bit map follows the header, and **nwds** gives the length of the map in 8-byte units. To save space, some zero bits at the end of the map are not backed up.

2.3.7.7 Location Headers

FS_DS headers have the following format:

```
struct {
   struct hdr h;
   char       nid[8];
   char       qdir[2];
};
```

The fields in this header are:

nid Node ID of the system being backed up. For local files, this field contains the node ID of the local system.

qdir Full path name of the directory that is being backed up, beginning at the root directory.

2.3.7.8 File Headers

FS_INODE and **FS_NAME** headers have similar formats:

```
struct {
   struct hdr h;
   ulong      ino;
   ulong      mode;
   ushort     nlink;
   ulong      uid;
   ulong      gid;
   off_t      size;
   time_t     atime;
   time_t     mtime;
   time_t     ctime;
   ushort     devmaj;
   ushort     devmin;
   ushort     rdevmaj;
   ushort     rdevmin;
   off_t      dsize;
   ulong      gen;
   ulong      cmtcnt;
   ulong      fstore;
   ulong      version;
   ushort     rdevsite;
   char       sbflag;
   char       pada;
   char       name[4];
};
```

The fields **mode** through **ctime** are based on the inode on disk.

Other fields are:

ino I-number of file.

devmaj,devmin Combined, these fields contain what is returned in **st_dev** by the **statx** system call for this file.

rdevmaj Major device number of this file (character- and block-special files only).

rdevmin Minor device number of this file (character- and block-special files only).

dsize Size of the file after backup. This differs from **size** if the file was compressed during backup.

gen Generation number of the inode which is a reuse count of the inode slot.

cmtcnt Used in TCF replicated file systems. The sequence number within the file system of the last commit to this file.

fstore Used in TCF replicated file systems. The storage attribute used in determining where copies of the file are to be stored.

version Version of the file; sequence count of the number of commits on this generation number; resets on the last

unlink.

rdevsite Device site number. TCF cluster site number to which this device is attached (character- and block-special files only).

name The null-terminated name of the file that is supplied by the user. This field is absent from **FS_INODE** headers.

FS_DINODE and **FS_DNAME** headers have similar formats. They are used by the **-c** flag of the **backup** command for compatibility with AIX/RT:

```
struct  {
    struct hdr h;
    ushort     ino;
    ushort     mode;
    ushort     nlink;
    ushort     uid;
    ushort     gid;
    off_t      size;
    time_t     atime;
    time_t     mtime;
    time_t     ctime;
    ushort     devmaj;
    ushort     devmin;
    ushort     rdevmaj;
    ushort     rdevmin;
    off_t      dsize;
    char       name[4];
};
```

2.3.7.9 End of Volume or Backup

FS_VOLEND and **FS_END** headers contain only the **hdr** structure.

2.3.7.10 Backup History

A backup history is kept in the **/etc/budate** file. The entries are in no particular order. Each entry has the following format:

```
struct {
     char    id_name[16];
     char    id_incno;
     time_t  id_budate;
};
```

The fields of each entry are:

id_name Name of the file system
id_incno Incremental level number (0-9)
id_budate Date of most recent backup of the file system at that level.

Related Information
In this book: "filesystems" in topic 2.3.18.

The **backup**, **pack**, and **restore** commands in *AIX Operating System Commands Reference*.

2.3.8 cc.cfg

Purpose
Defines values used by the C, FORTRAN, and Pascal compilers.

Description
The **cc.cfg** file defines values used by the **cc** program to run compilers. Normally, the **cc.cfg** file contains entries only for the C compiler provided with the system. Entries are made to this file to support C compilers for other systems as they are added.

Besides the default stanza, AIX provides the stanza "posix" as a way for application programs to compile programs written to be strictly conformant to the POSIX 1003.1 standard, as defined in the *Portable Operating System Interface for Computer Environments (POSIX), IEEE 1003.1-1988*. Applications compiled with "cc -F:posix" will be compiled with the symbol **_POSIX_SOURCE** defined. This symbol hides all symbols in POSIX-defined header files which are not defined by the POSIX 1003.1 standard. Compiling an application in this way will guarantee that the application doesn't make use of any AIX extensions and hence is a portable application. Additionally, the hiding of AIX extensions guarantees that these extensions will not conflict with variables, data types and other data structures which the application has defined. Consult the POSIX 1003.1 standard for details on how to write a strictly-conforming POSIX application.

This file is an attribute file. The name you specify when you run the **cc** program (it can be linked to several different names) determines which stanza of the **cc.cfg** file is used. Normally, the **cc** program runs as **cc**; therefore, the first stanza is almost always selected.

You can specify the following attributes:

as The path name to be used for the assembler.

asflags A string of values, separated by commas, to be passed to the assembler.

asopt A string naming optional flags that, if encountered on the **cc** command line, should be passed to the assembler. See description of the **cppopt** field.

ccom The path name to be used for the C compiler front end; this is the parser (**vsc**).

ccomflags A string of values, separated by commas, to be passed to the C compiler.

ccomopt A string naming optional flags that, if encountered on the **cc** command line, should be passed to the C compiler. See **cppopt**.

cform The path name to be used for the code formatter (**vspass3**).

Note: The code generator and the code formatter do not accept optional flags.

cgen The path name to be used for the code generator (**vspass2**).

cpp The path name to be used for the preprocessor.

cppflags The following C-preprocessor symbols are predefined for use by application development:

_AIX	Defined on all AIX platforms.
AIX	Defined on all AIX platforms.
u370	Defined on AIX/370 machines.
AIX370	Defined on AIX/370 machines.
i386	Defined on AIX PS/2 machines.
_I386	Defined on AIX/PS/2 machines.
NLS	Usable with NLS library routines.
HIGHC	Code conforms to the High C compiler standard.
_POSIX_SOURCE	Code conforms to POSIX standard.
STDC	Code conforms to ANSI C standard.

cppopt A string naming optional flags that, if encountered on the **cc** command line, should be passed to the preprocessor. The string is formatted for **getopt()** subroutine, as a concatenation of flag letters, with a letter followed by a : (colon) if the corresponding flag takes a parameter.

crt, mcrt The path name of the object file passed as the first parameter to the link editor. In the presence of the **-p** flag to **cc**, the **mcrt** value is used; otherwise the **crt** value is used. The defaults are **/lib/crt0.o** and **/lib/mcrt0.o**.

crtn The path name of the object file passed as the parameter after the library flags to the link editor. This object file is necessary whenever shared libraries are being linked. The default is **/lib/crtn.o**.

csuffix The suffix for **C** source programs, the default is **c**.

dis Pathname to use for the disassembler.

fcom The path name to be used for the FORTRAN compiler front end (**vsfort**).

fcomflags A string of values, separated by commas, to be passed to the FORTRAN compiler.

fcomopt A string naming optional flags that, if encountered on the **cc** command line, should be passed to the FORTRAN compiler.

flib The path name to be used for the FORTRAN Library.

fsuffix The suffix for VS FORTRAN source programs, the default is **f**.

hsuffix A second suffix for **C** source (enabled by using the **-h** flag to the **cc** command), the default is **h**.

hcpass1 First pass of the High C Compiler. This is the parser.

hcpass2 Second pass of the High C Compiler. This takes intermediate code generated by a High C Compiler and produces object code.

hcopt A string naming option flags that, if encountered on the cc command line, should be passed to the High C Compiler.

hcflags A string of values, separated by commas, to be passed on to the

High C Compiler.

hcansi Directory containing ANSI libraries. Used only if **-Hansi** flag is turned on.

hcinline Inline code pass of High C Compiler.

ld The path name to be used for the link editor.

ldflags A string of values, separated by commas, to be passed to the link editor. These are in addition to those implicitly provided as described in the **cc** command.

ldopt A string naming optional flags that, when encountered on the **cc** command line, to be passed to the link editor. See **cppopt**.

libraries Flags, separated by commas, to be passed as the last parameters to the link editor as the default libraries, the default is **-lrts,-lc**.

osuffix The suffix for object files, the default is **o**.

pcom The path name to be used for the Pascal compiler front end (**vspascal**).

pcomflags A string of values, separated by commas, to be passed to the Pascal compiler.

pcomopt A string naming optional flags that, if encountered on the **cc** command line, should be passed to the Pascal compiler.

plib The path name to be used for the Pascal Library.

psuffix The suffix for VS Pascal source programs, the default is **p**.

ssuffix The suffix for assembler programs, the default is **s**.

use Values for attributes are taken from the named stanza in addition to the local stanza. For single-valued attributes, values in the **use** stanza apply if no value is provided in the local stanza (or default stanza). For comma-separated lists, the values from the **use** stanza are added to the values from the local stanza.

Example

```
* CC configuration file:

default:
     cpp       = /lib/cpp

* standard cc
cc:
     use       = DEFLT
     crt       = /lib/crt1.o
     crtn      = /lib/crtn.o
     mcrt      = /lib/mcrt1.o
     libraries = -lrts,-lc

* common definitions
```

```
DEFLT:
     ccom      = /lib/vsc
     ccomopt   = ""
     fcom      = /lib/vsfort
     fcomopt   = ""
     pcom      = /lib/vspascal
     pcomopt   = ""


     cgen      = /lib/vspass2
     cform     = /lib/vspass3
     as        = /bin/as
     ld        = /bin/ld
     cppflags  = D_AIX,-Daiws,-DAIX,-DNLS
     ldflags   = -K,-T0x00400000

     hcpass1   = /lib/hc1com
     hcpass2   = /lib/hc2com
     hcopt     = " "
     hcflags   = " "
     hansi     = /lib
     hcinline  = /lib/inline
```

File
/etc/cc.cfg

Related Information
In this book: "getopt" in topic 1.2.106 and "attributes" in topic 2.3.5.

The **as**, **cc**, and **ld** commands in *AIX Operating System Commands Reference*.

2.3.9 connect.con

Purpose
Controls communication connections and data transfer.

Description
The connection configuration file, **/usr/lib/INnet/connect.con** or **$HOME/bin/connect.con**, controls the setup of connections for the **connect** program and for certain optional communications programs. It provides a very general, flexible mechanism to specify how connections are made and how data is transferred after making a connection.

The **connect.con** files are attribute files. The following attributes may appear in the connection control file.

Subtopics
2.3.9.1 Connection Options
2.3.9.2 Line Options and Parameters
2.3.9.3 System Options
2.3.9.4 Diagnostics
2.3.9.5 Login Script
2.3.9.6 Talker Program

2.3.9.1 Connection Options

The connection options and their descriptions are:

prefix, address, suffix
The telephone number to dial or the network address to contact. The actual number is constructed by concatenating the prefix (if any), the address, and the suffix (if any). Usually the **prefix** and **suffix** are defined in **/etc/ports** because they depend on the peculiarities of the dialer, and the **address** is defined in **connect.con**.

Multiple addresses can be specified by consecutive **address** assignment lines or by multiple **address** values separated by commas. The addresses are tried in the order given. To specify a comma as part of the command that is sent to the modem, enclose the entire **address** value in quotation marks.

connect Type of connection to make. This option is specified in **/etc/ports** since it is usually associated with the hardware configuration of the outgoing line. Permissible values are:

permanent The connection is hard-wired. No dialing or other special attention is needed.

manual The connection must be completed manually. This generally implies a modem that does not dial, for example, an acoustic coupler.

hayes_1200 The line has a Hayes Stack Smartmodem 1200.

hayes_2400 The line has a Hayes Stack Smartmodem 2400.

vadic The line has a Racal-Vadic 3451P autodialer.

ventel The line has a Ventel MD212+ autodialer.

other_name The line is associated with a dialer program, which is not built into the **connect** program. This option allows you to augment the capabilities of the **connect** program and other communications programs when dealing with new types of communications lines and dialers. The program searches for the named dialer program in **/usr/lib/INnet/dialers** or **$HOME/bin**.

The assumptions made for dialer programs you supply are: the port to be used can be opened before dialing and the file will be opened as descriptor 3. Two parameters are passed: number to dial as parameter 1, and dialer hardware to use or value of the dialer option, if any as parameter 2. Any code exit from the dialer except 0 indicates the dialer failed. The failure code returned by the dialer determines the message printed by the programs.

linetype Type of communication line protocols, either synchronous or asynchronous. Different protocols are used on different line types, so the talker programs may differ. The default linetype is asynchronous.

type The name invoked with the **connect** program that determines the kind of connection attempted. Only those stanzas with the proper type are processed. Currently, the **connect** program itself uses only **terminal** type stanzas. The default type is **terminal**.

use This option directs the **connect** program to read the named stanza and follow the instructions there.

2.3.9.2 Line Options and Parameters

Line options and parameters used are:

min The minimum value to use in kernel buffering. **Min** value characters must be received before a call to the **read** system call returns, unless value specified in **time** elapses.

parity The line is checked for the indicated parity: **even, odd, any,** or **none**.

speed The transmission speed, generally 110, 300, 1200, 2400, 9600, and so on.

time The value to use in kernel buffering. **Time** in tenths of a second to receive a character before a call to the **read** system call returns unless **min** characters are received. See the discussion of ICANON in "termio" in topic 2.5.28. Setting these parameters can result in improved performance.

timeout The time limit to complete the connection in seconds. When the time limit expires, the connection is aborted. This attribute is not needed for devices with a built-in timeout.

2.3.9.3 System Options

The system options are:

device The name of the special file to use to make the connection. The device must appear in **/etc/ports** (see "ports" in topic 2.3.46) and the information in the ports file entry that is made available to the **connect** program. Note that this attribute can appear only in the last of the list of stanzas associated with making the connection on this device, and that the **use** option must not appear.

dialer This option specifies the dialer hardware to be used in dialing the number. It is normally in **/etc/ports** file, associated with the device to be used. It may also be specified in a connection file, so that its value can be passed to a user-specified dialer program.

2.3.9.4 Diagnostics

The following diagnostics are displayed, based on the return value from system- or user-supplied dialer programs. The values 8 through 14 are treated as fatal errors.

Code	Message
0	Connected
1	Cannot open dialer
2	Busy or no answer
3	Not able to fork
4	Terminated attempts
5	Communication failure
6	Busy
7	No answer
8	Dead phone
9	Bad phone number
10	Cannot open device specified
11	Address not specified
12	Bad **connect.con** format
13	Cannot run dialer
14	No autodialer specified.

2.3.9.5 Login Script

A login script is file with the given name that is interpreted before notifying you that the connection is complete. Script files are located either in the **$HOME/bin** file or in the **/usr/lib/INnet/scripts** file.

script A script file is organized into a group of states. In each state, the script file optionally sends a string to the remote system, then waits for a response. Several possible responses can be listed for each state along with an action to be performed if the response is received. A time limit can also be set in each state, along with an action to be performed if the time expires without an expected string arriving. The actions are to terminate script interpretations, with either a success or failure indication, or to move to another state. A sample script is shown under "Example" in topic 2.3.9.6.

DONE
A successful termination of script interpretation.

ERROR string
An unsuccessful termination of script interpretation. The last message received from the remote site is reported to you.

GOTO n
Continues processing in state **n**.

RECV string action
This action is performed if the given string is received.

SEND string
Sends the given string to the remote system. Any name enclosed in braces in the string is taken to be an option reference and is replaced by the value of that option. If that option is not present in the list of stanzas, you are prompted for its value using the option name as the prompt. If a - (dash) precedes the name within the braces, the typed characters are not echoed. This is handy for including passwords as parameters in the script file without having them stored on the system. Thus, script files can be given parameters so that they can be used in different connection stanzas and by different users.

STATE n
Declares the beginning of state **n**.

TIMER n action
This action is performed if no expected string is received in **n** seconds.

2.3.9.6 Talker Program

A talker program handles the work of moving data across a connection. This program runs after a connection is established. The default talker for the **connect** program is **atalk**. You can override this and specify your own talker program.

talker This is the name of the program to run when the connection is made. The conventions observed between the **connect** program and the talker are not complex: the connection is opened by the program as file descriptor 3. The only flag passed by **connect** to the talker program is:

-llockfile

Note: If the **-l** flag is present, the talker must remove the named **lockfile** to make the port available to other users.

flags This option passes flags (other than the above) to the talker program. This option is valid with both default or user-specified talkers.

Example
A typical script might be:

```
       STATE 0   RECV User:              GOTO 1
                 TIMER 10                ERROR "No login"

       STATE 1   SEND "{myname}\n"
                 RECV Password:          GOTO 2
                 RECV "Unknown:"         ERROR "Name unknown"
                 TIMER 10                ERROR "No password msg"

       STATE 2   SEND "{-mypass}\n"
                 RECV "$"                DONE
                 RECV Invalid            ERROR "Wrong password"
                 TIMER 20                ERROR "No prompt"
```

This script could be used for login to a remote AIX system. In this file, **connect** waits up to 10 seconds for a **User:** prompt. When received, it sends the value of the **myname** option from the control file or by prompt, as the user name. It then waits for 10 seconds for the **Password:** prompt, then it sends the value of **mypass** as the password. The password is not echoed. It then waits another 20 seconds for another prompt. At each stage, it looks for messages that could occur if the given user name or password is invalid. With more states, you can write a script that tries several different user names and types the necessary information to dial through a port selector.

Files

/usr/lib/INnet/connect.con
$HOME/bin/connect.con

Related Information
In this book: "attributes" in topic 2.3.5, "ports" in topic 2.3.46, and "termio" in topic 2.5.28.

The **connect** and **uucp** commands in *AIX Operating System Commands Reference*.

2.3.10 core

Purpose
Contains an image of user memory at the time of an error.

Synopsis

```
#include <sys/b370/coredump.h>
#include <sys/i386/coredump.h>
#include <sys/reg.h>
```

Description
The system writes a memory image of a terminated process when various errors occur. See the **sigaction** system call for the list of errors. The most common are memory address violations, illegal instructions, bus errors, and user-generated quit signals. The memory image, called **core**, is written in the process working directory.

This memory image file is not written if any of the following are true:

The process has an effective user ID that is different from the rea user ID.

A file named **core** already exists in the current directory and is not writable by this user or has a link count greater than 1 (one).

The user does not have write permission in the current directory whic is required to create the memory image file.

The memory image file resides in a system replicated file system, suc as the root file system. If it is important to create a memory image file for a daemon or other program which has its current directory in a directory such as "/", temporarily create a symbolic link named **core** which points to a file in another file system.

The memory image file contains common sorts of information about processes running on AIX/370 and AIX PS/2; however, the files are binary data files and do need to be examined from the same type of machine on which they were created. Use the **file** command to determine which machine type created a memory image file. Use a user-level debugger such as **dbx** to interpret a memory image file.

The first section of the memory image is a header containing parts of the system's per-user data for the process, including the registers as they were at the time of the error. It also contains an array of structures describing the layout of the rest of the memory image file. The remainder of the memory image file represents the actual contents of the user-writable portions of the process when the memory image was written. Read-only portions of the memory image are not dumped.

The format of the information in the first section of the memory image (the header) is described by the **corehdr** structure, which is defined in the files **/usr/include/sys/i386/coredump.h** and **/usr/include/sys/b370/coredump.h**. The general purpose registers are in the **cd_regs** array as described within that file. The floating-point registers are in an **fp87state** structure defined in **/usr/include/sys/regr.h**.

The process's memory image is written in segments. The header contains an array of up to **MAX_CORE_SEGS dumpseg** structures which describe the

segments of the process (including those which are not dumped). If the segment was dumped, **cs_offset** is the segment's offset into the memory image file; otherwise, **cs_offset** is 0. The field **cs_len** is the length of the segment in bytes and the field **cs_address** is the virtual address of the segment in the process. The field **cs_type** describes the type of segment. The above structures, their fields, and the possible values of **cs_type** are described in the files **/usr/include/sys/i386/coredump.h** and **/usr/include/sys/b370/coredump.h**.

File

core

Related Information

In this book: "setuid, setgid" in topic 1.2.255 and "sigaction, sigvec, signal" in topic 1.2.263.

The **dbx** command in *AIX Operating System Commands Reference*.

2.3.11 cpio

Purpose
Describes copy in and out (**cpio**) archive file.

Description
When the **-c** flag of the **cpio** command is not used, the header structure is:

```
struct {
  short
     h_magic,
     h_dev;
  unsigned short
     h_ino,
     h_mode,
     h_uid,
     h_gid;
  short
     h_nlink,
     h_rdev,
     h_mtime[2],
     h_namesize,
     h_filesize[2];
  char
     h_name[n];         /* described below */
} Hdr;
```

When the **cpio** command is used with the **-c** flag, the header for the **cpio** structure may be read as:

```
sscanf(Chdr,"%6ho%6ho%6ho%6ho%6ho%6ho%6ho%6ho%11lo%6ho%11lo%s",
&Hdr.h_magic, &Hdr.h_dev, &Hdr.h_ino, &Hdr.h_mode,
&Hdr.h_uid, &Hdr.h_gid, &Hdr.h_nlink, &Hdr.h_rdev,
&Longtime, &Hdr.h_namesize, &Longfile, &Hdr.h_name);
```

Longtime and **Longfile** are equivalent to **Hdr.h_mtime** and **Hdr.h_filesize**, respectively. The contents of each file together with other items describing the file are recorded in an element of the array of varying length structures. The member **h_magic** contains the constant octal 070707 (or 0x71c7). The **stat** system call explains the meaning of structure members **h_dev** through **h_mtime**. The length of the null-terminated path name, **h_name**, including the null byte is indicated by **n**, where **n** = (**h_namesize % 2) + h_namesize**. In other words, **n** is equal to **h_namesize** if **h_namesize** is even. If **h_namesize** is odd, **n** is equal to **h_namesize** + 1.

The last record of the archive always contains the name **TRAILER!!!**. Directories and the trailer are recorded with **h_filesize** equal to 0.

Directories and special files are treated in a slightly different way. A directory is written with a 0 size, meaning no data blocks follow. Also, if the directory is a hidden directory, an @ (at) is appended to the name. A special file size is also 0; its major/minor device number is stored in the **h_rdev** field and its device site number is stored in the **h_namesize** field.

Related Information
In this book: "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf, wsscanf" in topic 1.2.241 and "statx, fstatx, stat, fstat, fullstat,

ffullstat, lstat" in topic 1.2.282.

The **cpio** and **find** commands in *AIX Operating System Commands Reference*.

2.3.12 .cshrc, .login

Purpose
Sets the **csh** user environment at login time.

Description
The **.cshrc** file provides commands to be executed at login time and variable assignments to be set and exported into the environment. The **.cshrc** file is read every time the **csh** is started, while the **.login** file is read only on start up.

After the login program adds the LOGNAME (login name) and HOME (login directory) parameters to the environment and starts **csh**, the commands in **$HOME/.cshrc** are executed, followed by the commands in **$HOME/.login**, if they are present. The **.cshrc** and **.login** files are used to tailor the user environment variables. The **.cshrc** file is often used to set exported environment variables and terminal modes. The system administrator can use **adduser** to set default **.cshrc** and **.login** files in each user home directory. Users can tailor their environment as desired by modifying their **.cshrc** and **.login** files.

Files

$HOME/.cshrc
$HOME/.login

Related Information
In this book: "environment" in topic 2.4.6 and "TERM" in topic 2.4.26.

The **env**, **login**, **mail**, **csh**, and **stty** commands in the *AIX Operating System Commands Reference*.

2.3.13 ddi

Purpose
Contains device-dependent information (**ddi**).

Description
A **ddi** file contains information for customizing classes (or types) of devices supported by the system. The information in this file may be modified using the **devices** command or an editor program. The **ddi** files are attribute files that are located in the **/etc/ddi** directory. See "attributes" in topic 2.3.5 for the format of attribute files.

The equivalent of a **ddi** file can be created and added to the system. ***Customize helper*** programs convert the parameters in the files into a format required by the operating system device drivers. A **ddi** file contains the following information:

Device-dependent information. This is a series of keywords whos values the user supplies when the device is defined.

Instructions to the customize helper program for processing inpu parameters.

Mapping information for the **ddi** structure.

Bit field mapping information

Devices that can be added, deleted, or changed by **devices** must have a **ddi** file.

Note: In the multibyte environment, the **ddi** file can contain only ASCII characters.

Subtopics
2.3.13.1 Keywords

2.3.13.1 Keywords

The following keywords are used in the stanzas of device-dependent information files. These keywords describe attributes and settings for a particular device that may be changed to suit your device.

Miscellaneous Keywords

Key Word	Description	Possible Choices
sysadd	Specifies the action that the **devices** command takes after adding the device. The valid choices are: **a** Rebuilds the kernel and IPLs the system **o** Runs the **osconfig** command **none** Takes no special action.	a, o, none
sysdel	Specifies the action that the **devices** takes after deleting the device. The valid choices are: **a** Rebuilds the kernel and IPLs the system **o** Runs the **osconfig** command **none** Takes no special action.	a, o, none

Printers and Plotters: Keywords followed by an asterisk (*) can be changed only when adding or changing information about a non-IBM printer.

Key Word	Description	Possible Choices
aa	**Automatic Answering**: Does the printer support communication auto answering?	
alf	**Automatic Line Feed**: Does the printer have automatic line feed with carriage return.	
ars*	**Aspect Ratio Support**: Does the printer have a "Set Aspect Ratio" control?	yes, no
backs*	**Backspace Support**: Does the printer have the ability to backspace (move print head backward while printing a line)?	yes, no
bigs	**Bit-Image Graphics Support**:	yes, no

	Does the printer have bit image graphics controls?	
bm	**Bottom Margin (last line)**: Refers to last line of text at bottom of a page; for instance, to leave a one-inch bottom margin on a page 66 lines long, you might set the margin on line 60. The value is determined by multiplying the length of the page in inches by the number of lines per inch.	1 - [length(in.) x lines/in.]
caps	**Map Lowercase Alphanumerics**: Maps lowercase alphanumerics to uppercase.	yes, no
cdp	**Condensed Print**: Should a file be printed with condensed print?	yes, no
cdpg	**Code Page**: Specifies the code page loaded into the printer.	437 (PC), 850 (MLP)
colp*	**Color Printer**: Is the printer capable of printing in color?	yes, no
cps*	**Condensed Print Support**: Does the printer support printing in condensed characters?	yes, no
cr	**Color Ribbon**: Is the printer capable of using a color ribbon?	yes, no
cs	**Character Set**: Refers to the specific character set to be used for printing.	1, 2
cus*	**Continuous Underscore Support**: Is the printer capable of underscoring characters?	yes, no
dpc	**Default Print Color**: Refers to the color to use for printing when a file doesn't contain codes that specify a color: usually black, blue, red, or yellow.	black, blue, red, yellow
dsp	**Double Strike Print**: Should double-strike be turned on?	yes, no
dsps*	**Double Strike Print Support**: Does the printer have a control to double-strike characters and provide boldface?	yes, no

dwp	**Double Width Print**: Should a file be printed with a double-width character set?	yes, no
dwps*	**Double Width Print Support**: Does the printer have the ability to print with a double-width character set?	yes, no
ep	**Emphasized Print**: Should emphasized print be turned on? Every character is overstruck with a second pass of the print head.	yes, no
eps*	**Emphasized Print Support**: Does the printer have a control to do emphasized print?	yes, no
fid	**Font ID**: ID of the font used by the printer.	11
fl	**Form (page) Length**: Refers to the length of the paper in terms of the number of lines per page. The value is determined by multiplying the length of paper (in inches) by the number of lines printed per inch.	1 - [length (in.) x lines/in.]
fw	**Form Width (right margin)**: Refers to the width of paper in terms of the number of characters per line. The value is determined by multiplying the width of the paper (in inches) by the number of characters printed per inch).	1 - [width(in.) x pitch]
hsi*	**Horizontal Spacing Increment**: What horizontal increment is used in the ESC K control?	60, 70
hts*	**Horizontal Tab Support**: Does the printer have horizontal tab controls?	yes, no
htvi	**Text Vertical Increment**: The vertical index increment used by subsequent CUU (ESC A) multibyte controls. (See "Multi-Byte Controls" in topic 2.4.3.3.2.)	72
ip	**Initialize Printer**: Refers to the initial state of the	true, false

	printer after power is applied.	
js*	**Justification Support**: Does the printer support an even right margin?	yes, no
kpoe	**Keep Printing on Error**: Should the printer complete the print job despite errors (without sending an error message to the user)?	yes, no
lm	**Left Margin**: Refers to the area on a page between the left edge and the leftmost character position on the page.	0 - [width(in.) x pitch]
lpi	**Lines Per Inch**: Refers to the number of print lines per inch, to line spacing density, and to the distance paper moves during a line feed.	6, 8
lrmc*	**Left/Right Margin Controls**: Does the printer have the ability to change left and right margins (does it have left and right margin control codes)?	yes, no
mccs*	**Multibyte Control Code Support**: Does the printer support IBM/OEM multibyte controls? Or does the printer act like an Epson (5152)?	yes, no
nocr	**No Carriage Returns**: Substitute line feeds for carriage returns.	yes, no
noff	**No Form Feed**: Simulate the form feed function.	yes, no
pacs*	**Print All Characters Support**: Does the printer support ESC and ESC- controls?	yes, no
ph	**Paper Handling**: Refers to the ways the printer handles different types of paper. The manual-feed printer stops at the end of each page and waits for the user to insert another sheet and press the start button. A printer with an automatic sheet-feed mechanism feeds paper to the printer.	0=manual; 1=automatic; 2=continuous (continuous form paper)

pitch	**Character Pitch**: Refers to the number of characters per linear inch; for instance, 10-pitch type has 10 characters per inch.	10, 12, 15
plot	**Pass Data Directly to Device Without Modification**: Overrides NOFF, NONL, NOTAB, NOBS, CAPS, and WRAP.	yes, no
pq	**Print Quality**: May select (on some printers) degrees of print quality: dp (for fast, low quality), text (for better draft quality), letter (for high-quality final text).	dp, text, letter
prin	**Printer Type**:	0,1,2,3,4,5,6,7,8
	0 = 5152	
	1 = 5182	
	2 = 3812	
	3 = 3852	
	4 = 5201	
	5 = 4201	
	6 = 4202	
	7 = 3852	
	8 = 5202	
psd	**Paper Source Drawer**: Refers to the location of the paper drawer from which paper is drawn for printing.	1=top; 2=bottom
pss*	**Proportional Spacing Support**: Does the printer support proportionally spaced printing?	yes, no
ptime	**Printer Timeout**: Specifies the printer timeout. Value is in seconds.	0-32768
rlfs*	**Reverse Line Feed Support**: Does the printer support the ESC J control?	yes, no
slap	**Skip Lines at Perforation**: Refers to the number of lines skipped at page breaks. The number is divided by 2, so	0-[length(in.) x lines/in.]

	that half the blank lines appear at the bottom of one page and half at the top of the next.	
sp	**Select Printer**.	true, false
sss*	**Superscript/Subscript Support**: Does the printer have the ability to print in superscript and subscript mode?	yes, no
tbc	**Transmit Buffer Control**: Number of bytes to buffer for transmitter.	0x00 - 0x10
tm	**Top Margin**: Refers to the number of lines to be skipped at the top of a page before printing begins. If the user specifies 6 lines, the first print line will be line 7. The value is determined by the length of paper (in inches) multiplied by the number of lines per inch.	0 - [length(in.) x lines/in.]
urpim	**User to Receive Printer Intervention Messages**: Refers to whether printer intervention messages are sent to any valid user or to the user who queued the print job.	Any user ID, pjo=Print Job Owner
vhs*	**Variable Horizontal Spacing**: Does the printer have ESC d and ESC e controls?	yes, no
vpqs*	**Variable Print Quality Support**: Does the printer have the ability to print different degrees of quality?	yes, no
vsi*	**Vertical Spacing Increment**: Refers to parts of inch supported in ESC J and ESC 3 control.	216, 144
vts*	**Vertical Tab Support**: Does the printer support vertical tabs?	yes, no
wll	**Wrap Long Lines**: Does the printer "wrap" lines? That is, will it break at the right margin those lines longer than specified form width and print the remainder on the next line?	yes, no

12ps*	**12 Pitch Support**: Does the printer support the printing of 12 characters per inch?	yes, no

Fonts: You can select any of the fonts or print type styles included on the font diskette. You must type the name of the font and the type style exactly as they appear on the diskette.

IBM 5201 Quietwriter

Key Word	Description	Possible Choices
cp1	**Code Page 1**	PC, A, B, C, D
fnt1	**Font 1**	
pitch1	**Character Pitch 1**	
type1	**Typestyle 1**	

IBM Proprinter (4201) and IBM Proprinter XL (4202): Four different sets of values are available for the IBM 4202 Proprinter (character pitch 1-4 and code page 1-4). Each set contains a designation for character pitch and code page. The possible choices for each of the four sets are shown below.

The code page choices for the IBM 4202 are the **PC** or multilingual code page (**MLP**). The **PC** code page contains the standard U.S. English alphabet, symbols, and punctuation marks. The **MLP** code page contains characters, symbols, and punctuation marks from many alphabets and languages.

Key Word	Description	Possible Choices
cp1	**Code Page 1**: On the American version of the 4201, **cp1** must be set to PC.	PC, MLP
pitch	**Character Pitch**	10, 12

TTY, TTYN, or TTYP Devices

Key Word	Description	Possible Choices
aa	**Automatic Answering**: Specifies whether the device supports communication auto answering. If the **aa** keyword is true, the **getty** command will transmit a modem command sequence to place the modem in automatic answering mode.	true, false
ae	**Automatic Enable**: At boot time,	true=enabled; false=not

	this determines the original enabling of the port. When **ae** is true, the port will be enabled whenever the system is ipl'ed. When **ae** is false, the port will be disabled. Setting **ae** to share or delay will result in the port being enabled with the proper locking to permit outgoing calls to originate from the PS/2.	enabled For TTY devices only: share=shared/bidirectional use; delay=delay logon herald
bpc	**Bits Per Character**: The number of bits per character used to transmit data from the PS/2 to the terminal or modem. This is usually set to 7 or 8 and must match whatever the terminal or modem is setup to use. NLS support requires a **bpc** value of 8.	5, 6, 7, 8
dvam	**Device Attachment Method**: Refers to whether the device is attached locally without a modem or remotely through a modem. Locally attached devices can be opened no matter what the state of the Carrier Detect, Clear To Send, Data Set Ready, and Ring Indicate signals. Opens do not complete for remote devices until Carrier Detect, Clear to Send, and Data Set Ready are all asserted.	0=local; 1=remote (modem)
ixp	**Include Xon/Xoff Protocol**: When **ixp** is set to true, Xon/Xoff flow control is used on both the received and transmitted data streams. The values of the **roffv**, **ronv**, **toffv**, and **tonv** keywords determine the actual characters which will be used to implement the flow control.	true, false
elevel	**Run-level of the getty process**: Each **getty** process spawned by **init** is assigned to a run-level in which it is allowed to exist. For example, if elevel is set to 14, then the **getty** is executed when the system is in multi-user mode. Refer to the **level** attribute in the description of **inittab** for more information.	0-6, a, b, c
logger	**Pty Supports Login Shell**: True indicates that a stanza for this **pty** will be put in the **/etc/ports** file that will allow the port to be enabled and disabled.	true, false

nosb	**Number of Stop Bits**: Refers to the number of stop bits used to frame each data character transmitted. The value chosen must be compatible with the **bpc** selection. 1.5 stop bits is only valid when **bpc** is set to 5.	1, 1.5, 2
pro	**Protocol**: Refers to communication protocol which determines how the modem control lines are used during a communications session. The **pro** keyword is generally set to **dtr**. The **dc** (Direct Control) value allows attachment of devices that use hard-wired flow control, such as certain serial printers and plotters. **Note:** While hard-wired flow control is sometimes referred to as DTR pacing, the **pro** keyword must be set to **dc** to support this function.	dtr, dc
pt	**Parity Type**: Specifies what type of parity, if any, the transmitted data will have. Received data will be checked to ensure it has the proper parity. The Native Serial Ports do not support the mark or space parity.	even, odd, mark, space, none
rts	**Receive/Transmit Speed**: Refers to the communication baud rate.	50, 75, 110, 134.5, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600, 19200
tt	**Terminal Type**: Refers to the type of device attached.	

3270 Devices

Key Word	**Description**	**Possible Choices**
lobibp	**Buffer Length**: Length of the buffers in the 3270 device driver buffer pool.	4096-65453
serial	**Serial Number**: PS/2 serial number, must be 7 digits.	0000001-9999999
machtype	**PS/2 Machine Type**.	8580, 8570
mnonid	**Number of 3270 Sessions**:	1-8

	There may be up to eight simultaneous Distributed Function Terminal sessions per 3270 Connection Adapter.	
slow	**Slow Device Mask Enables or Disables Slow Device on a per 3270 Session Basis**: If **slow** device is set to 2, then 3270 session will be set to a **slow** device. If **slow** is set to 28, then 3270 sessions 2 and 8 will be **slow** devices.	0-12345678
printer	**Printer Device Mask**: Refer to **slow** for interpretation of mask.	0-12345678

Files

/etc/ddi/diskette
/etc/ddi/ethernet
/etc/ddi/token
/etc/ddi/opprinter
/etc/ddi/plotter
/etc/ddi/pprinter
/etc/ddi/sprinter
/etc/ddi/tty
/etc/ddi/pty
/etc/ddi/tape
/etc/ddi/c327
/etc/ddi/nty

and possibly others.

Related Information
In this book: "attributes" in topic 2.3.5, "descriptions" in topic 2.3.14, "kaf" in topic 2.3.30, "master" in topic 2.3.32, "options" in topic 2.3.43, "predefined" in topic 2.3.47, and "system" in topic 2.3.56.

2.3.14 descriptions

Purpose
Describes the meaning of **ddi** file keywords.

Description
The **/etc/ddi/descriptions** file contains a sorted list of descriptions for each of the keywords used in **ddi** files. The **devices** command uses this file to explain the meanings of the keywords during the **add**, **change**, and **showdev** subcommands.

The **/etc/ddi/descriptions** file must be sorted by keyword, and each line must follow the following format:

keyworddescription

where:

keyword Names a keyword that is used in a **ddi** file. This field is exactly 10 characters long, is padded on the right with spaces, and contains no tabs.

description Describes the meaning of the keyword. This field is exactly 28 characters long, is padded on the right with spaces, and contains no tab characters.

Note: The **/etc/ddi/descriptions** file must be sorted alphabetically by the **keyword** field. If it is not sorted, then the **devices** commands displays incorrect information about the meanings of keywords.

The use of extended characters in the **/etc/ddi/descriptions** file is not supported.

File

/etc/ddi/descriptions

Related Information
In this book: "ddi" in topic 2.3.13 and "options" in topic 2.3.43.

2.3.15 devinfo

Purpose
Contains device characteristics.

Synopsis
#include <sys/devinfo.h>

Description
The **devinfo** structure is defined for each device. The **IOCINFO** operation of the **ioctl** system call fills in this structure. The information returned by a device varies. Most devices, other than disk devices, return a **devtype** value and the remainder of this structure contains zeros. This structure provides information about the capabilities of a device, rather than its current status or settings. For example, types of information provided are the number of characters a printer handles per line or the diskette capacity in number of blocks.

```
struct devinfo
{  char devtype;
   char flags;
   union
   { struct        /* for disks */
     {short  bytpsec;             /* bytes per sector */
         short  secptrk;          /* sectors per track */
         short  trkpcyl;          /* tracks per cylinder */
         long   numblks;          /* blocks this partition */
       } dk;
       struct     /* for memory mapped displays */
      { char  capab;              /* capabilities */
         char  mode;              /* current mode */
         short  hres;             /* horizontal resolution */
         short  vres;             /* vertical resolution */
      } tt;
      struct       /* for ethernet interfaces */
      {  unsigned short capab;    /* capabilities */
         char haddr[6];           /* hardware address */
      } en;
      struct       /* for block i/o device */
      {
        struct
         {
          char  type;             /* hardware type: ethernet or token rin
          char  if_flags;         /* up/down  1=ATTACHED
                                              2=RUNNING
                                              3=PRIMARY INTERFACE */
          char  haddr[6];      /* hardware address: ethernet or token rin
          long  mymach;           /* local IP address */
          long  subnet_mask;      /* subnet mask */
          int   mtu;              /* maximum transmission unit */
          char  if_name[IFNAMSIZ];/* name of interface */
         } lan[MAXIFS];
      } bio;
      struct        /* for magnetic tapes */


      { short       type;         /* what flavor of tape */
                                  /* defined below */
      } mt;
      struct        /* for mouse */
```

```
        {    short      m_xres;      /* best x resolution (points/cm) */
             short      m_yres;      /* best y resolution (points/cm) */
             short      pad;         /* make it for both 286 & 386 procs */
       } mo;
      } un;
  };
```

The following flags specify some generic capabilities (See DD_DISK) :

Constant	Value	Function
DF_FIXED	01	Non-removable
DF_RAND	02	Random access possible
DF_FAST	04	A relative term

The following flags are System/370 only:

Constant	Value	Function
DF_FBA	0x80	Fixed-block-architecture disk (3310/3370)
DF_CKD	0x40	Count-key-data disk (3310/3350/3375/3380)
DF_BLKIO	0xc0	Bulk-I/O disk
DF_MASK	0xc0	Mask for disk type

The following flags are 386-only:

Constant	Value	Function
DF_ESDI	0x20	Enhanced small device interface
DF_ST506	0x10	Seagate Technology 506 interface
DF_I386_MASK	0x30	Mask for disk type

The **devinfo** structures are defined for the following devices (specified in the **devtype** file) :

DD_DISK Indicates a disk. This **devtype** is **R**. The driver determines the value. The fixed disk has flags **DF_RAND|DF_FIXED|DF_FAST**, while the diskette has flag **DF_RAND** (see "fd" in topic 2.5.9 and "hd" in topic 2.5.10).

The number of the bytes per sector, sectors per track, and tracks per cylinder for the fixed disk are predetermined. The minidisk table determines the number of blocks. For the diskette, the minor device driver or the physical media determines this information when the device is opened.

DD_LP Indicates a line printer. The **devtype** is **1**. This fills in the **devtype** field and returns zeros for the rest of the structure.

DD_PSEU Indicates a pseudo-device. The **devtype** is **Z**.

DD_TAPE Indicates a magnetic tape. The **devtype** is **M**.

DD_TTY Indicates a terminal. This returns a **devtype** of **t** and zeros for the rest of the structure.

DD_PUN Indicates a card punch. The **devtype** is **p**.

DD_RDR Indicates a card reader. The **devtype** is **r**.

DD_NET Indicates a network. The **devtype** is **N**.

DD_EN	Indicates an Ethernet interface. The **devtype** is **E**.
DD_EM78	Indicates a 3278/79 emulator. The **devtype** is **e**.
DD_TR	Indicates a Token Ring interface. The **devtype** is **t**.
DD_BIO	Indicates a block I/O device. The **devtype** is **B**.
DD_MOUS	Indicates a mouse. The **devtype** is **m**.
DD_OSM	Indicates an **OSM**. The **devtype** is **o**.
DD_VM	Indicates a virtual machine command interface. The **devtype** is **o**.
DD_MEM	Indicates an interface to kernel memory. The **devtype** is **k**.
DT_STREAM	Indicates a streaming tape drive. The **devtype** is **l**.
DT_STRTSTP	Indicates a start-stop tape drive. The **devtype** is **2**.

Related Information
In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "fd" in topic 2.5.9, and "hd" in topic 2.5.10.

2.3.16 dir

Purpose
Describes the format of a directory.

Synopsis

```
#include <sys/types.h>
#include <dirent.h>
```

Description
A directory is a file that a user is not allowed to write into directly. A directory file contains a variable-sized entry for each file in it. A bit in the file mode of the inode entry indicates that the corresponding file should be treated as a directory. For additional information about a system volume format, see "fs" in topic 2.3.20 and "inode" in topic 2.3.29.

The recommended way for a program to read a directory is through **readdir** (see "directory: opendir, readdir, telldir, seekdir, rewinddir, closedir" in topic 1.2.60) or **readx** (see "read, readv, readx" in topic 1.2.224). If a program uses **read**, and that process hasn't been marked with the "real directory read" attribute (see "ulimit" in topic 1.2.313), the system will presume that the program is expecting fixed-sized directory entries. See System V compatibility, described below.

A directory consists of some number of blocks of **DIRBLKSIZ** bytes (defined in **<dirent.h>**), or exactly one block of size **SMBLKSZ** bytes (defined in **<sys/ino.h>**). Each directory block contains some number of directory entry structures, which are of variable length. Each directory entry has a **struct dirent** at the front of it, containing its inode number, the length of the entry, and the length of the name contained in the entry. These are followed by the name padded to a 16-byte boundary with null bytes. All names are guaranteed null-terminated. The maximum length of a name in a directory is **NAME_MAX**. The maximum length of an entire path name is **PATH_MAX**. Neither **NAME_MAX** nor **PATH_MAX** includes the terminating null character.

The macro **NDIRSIZ(dp)** gives the amount of space required to represent a directory entry. Free space in a directory is represented by entries that have **dp->d_reclen > DIRSIZ(dp)**. All **DIRBLKSIZ** (or **SMBLKSZ**) bytes in a directory block are claimed by the directory entries. This usually results in the last entry in a directory having a large **dp->d_reclen**. When entries are deleted from a directory, the space is returned to the previous entry in the same directory block by increasing its **dp->d_reclen**. If the first entry of a directory block is freed, then its **dp->d_ino** is set to 0. Entries other than the first in a directory block do not normally have **dp->d_ino** set to 0.

```
struct  dirent  {
        ino_t      d_ino;
        u_short      d_reclen;
        u_short      d_namlen;
        char          d_name[NAME_MAX  + 1];
        /* typically shorter */
};

#define NDIRSIZ(dp) \
    (((sizeof(struct dirent) - (NAME_MAX+1)) + ((dp)->d_namlen+1 + 15)) &¬ 1
```

By convention, the first two entries in each directory are for . (dot) and .. (dot dot). The first . is an entry for the directory itself. The .. entry is for the parent directory. The meaning of the .. entry for the root directory of the master file system is modified. There is no parent directory; therefore, the .. entry has the same meaning as the . entry.

Subtopics
2.3.16.1 Compatibility Interfaces

2.3.16.1 Compatibility Interfaces

For compatibility with UNIX System V and earlier AIX systems, which employed fixed-sized 16-byte directory entries, the **direct** structure has been preserved. This is the structure returned by programs which call the **read** system call directly:

```
#define  DIRSIZ  14
struct   direct
{
         s_ino_t  d_ino;
         char     d_name[DIRSIZ];
};
```

For compatibility with 4.3BSD, which uses **struct direct** and **DIRSIZ** to describe the variable-sized directory entries, the following definitions are made. One must define **_BSD** in a program to get these 4.3BSD definitions.

```
#ifdef _BSD
#define direct dirent
#define DIRSIZ(dp) NDIRSIZ(dp)
#endif
```

Related Information
In this book: "read, readv, readx" in topic 1.2.224, "ulimit" in topic 1.2.313, "fs" in topic 2.3.20, and "inode" in topic 2.3.29.

2.3.17 errfile

Purpose
Contains system event log.

Synopsis
#include <sys/erec.h>

Description
When a system event occurs and logging is active, it generates an event record and passes the record to the event-logging daemon to be recorded in the event log. The **/etc/rasconf** file specifies the files where the events are to be logged. The default event log file is **/usr/adm/ras/errfile**.

Every record has a header. See "error" in topic 2.5.7 for the structure of a header. Each type of event record has its own format. The **/usr/include/sys/erec.h** file shows the format of the events currently logged. The error daemon process gathers the records from memory and writes them in the files on disk. The event log file is opened (if existing) or created. Next, the process opens the **/dev/error** special file, formats and writes the non-volatile random access memory (NVRAM), which can contain up to 16 bytes of information, and reads the events logged in memory. An analysis routine is called before an event is written to the **errfile**. For an error, this routine returns a buffer of probable cause information to aid in problem determination. This buffer is appended to the error entry, the length of the entry is adjusted, and then the entire entry is written to the file.

Some records in the event file are administrative. These include the startup record entered when logging is activated, the stop record written if the daemon is terminated gracefully, and the time-change record that accounts for changes in the system time of day.

Files

/usr/adm/ras/errfile
/dev/error
/etc/rasconf

Related Information
In this book: "rasconf" in topic 2.3.50 and "error" in topic 2.5.7.

The **errdemon** in *AIX Operating System Commands Reference*.

2.3.18 filesystems

Purpose
Centralizes file system characteristics.

Description
A file system is a complete directory structure, including a root directory and any directories and files beneath it. A file system is confined to a single partition. All of the information about the file system is centralized in the **filesystems** file. Most of the file system maintenance commands take their defaults from this file. The file is organized into stanzas whose names are file system names and whose contents are attribute-value pairs specifying characteristics of the file system.

The **filesystems** file serves two purposes:

It documents the layout characteristics of the file systems

It frees the person who sets up the file system from having to ente and remember items such as the device where the file system resides because this information is defined in the file.

If the Transparent Computing Facility is installed, there is a unique **/etc/filesystems** for each cluster site (**/etc/filesystems** is a symbolic link into the local file system).

Subtopics
2.3.18.1 File System Attributes

2.3.18.1 File System Attributes

Each stanza names the directory (which must be the full pathname) where the file system is normally mounted. The attributes specify all of the parameters of the file system. See "attributes" in topic 2.3.5 for the format of an attribute file. The attributes currently used are:

account Used by the **dodisk** command to determine the file systems to be processed by the accounting system. This value can be either TRUE or FALSE.

backupden Used by the **backup** command to determine the density of the default backup device associated with each file system. Density is measured in bytes per inch. The parameter is ignored for diskettes.

backupdev Used by the **backup** and **restore** commands to determine the default output device associated with each file system. The value of this keyword is usually the name of a diskette or magnetic tape special file.

backuplen Used by the **backup** command to determine the size of the default backup device associated with each file system. The size of a tape is measured in tracks times feet. For example, the **backuplen** for a 300-foot 9-track tape is 2700. This parameter is ignored for diskettes.

backuplev Used by the **backup** command to determine the default backup level to take for each file system. Backup levels are discussed in the **backup** command.

bad List of physically flawed disk blocks which are excluded from the pool of available file system blocks.

boot Used by the **mkfs** command to initialize the boot block of a new file system. This specifies the name of the load module to be placed into the first block of the file system.

check Used by the **fsck** command to determine the default file systems to be checked. TRUE enables checking while FALSE disables checking. If a number, rather than TRUE is specified, the number indicates which of multiple concurrent **fsck** processes will check this file system. This parallel checking, described in **fsck** command in *AIX Operating System Commands Reference*, permits multiple file systems to be checked in parallel when multiple drives exist.

cyl Used by the **mkfs** command to initialize the free list and super block of a new file system. The value is the number of blocks in one cylinder. It defines the size of an interleave cluster.

dev Identifies, for local mounts, the block special file where the file system resides. System management utilities use this attribute to map file system names to the corresponding device names. For NFS mounts, the *host:NFSdir* form is used. Host specifies the host machine on which the remote file system resides. *NFSdir* specifies the path name of the remote file system being mounted. Use ASCII characters for filesystem name to ensure successful communication across

different locales.

free Used by the **df** command to determine which file systems are to have their free space displayed by default. This value is either TRUE or FALSE.

freq Used by the **dumpbsd** command, **freq** specifies dump frequency, indicated by number of days.

ftype File system type. May include one of the following values:

- **nfs** An NFS mounted file system.
- **nonrepl** A standard file system which is not replicated on other sites. This filesystem is accessible by other sites in a TCF cluster only when the machine storing it is up.
- **repl** A filesystem which is one instance of multiple copies throughout a TCF cluster. The following value is used to differentiate between system and user replicated file systems:
  - **user** This filesystem is user replicated. If this value is not present, the file system is system replicated.

  One of the following may also be present:
  - **primary** This is the primary copy of the replicated file system, readable and writable by all sites in the TCF cluster.
  - **backbone** A backup copy of a replicated filesystem, readable by all sites, but writable only by the primary site.

  System replicated filesystems which are not primary or backbone must include the following:
  - **fstore** An fstore value found in /ect/fstore which indicates what files are present on this secondary copy.

gfs Global file system number. Used by TCF. Every minidisk on a node has a different gfs number.

gfspack Global file system. Always 1 for nonreplicated file systems. Used by TCF.

inodes Used by the **mkfs** command for reference and to build the file system. The value is the number of inodes (files) in the file system. If this attribute is not specified, the value is calculated from the size attribute.

mode Used by the **dumpbsd** command, **mode** specifies the type of file system desired, as follows:

rw Read/write device

ro Read only device

sw Swap device

nm File system normally not mounted

xx Ignore type.

mount Used by the **mount** command to determine whether or not this file system should be mounted by default. The possible values of **mount** are:

automatic Automatically mounts a filesystem when the system is rebooted. For example, in the sample file, the root file system is **mount=automatic**. This means the root file system mounts automatically when the system is rebooted. The TRUE value is not used so that **mount all** does not try to mount it. Also, the value is not FALSE because certain utilities, such as **ncheck** normally avoid file systems with a value of **mount=false**.

false This file system is not mounted by default.

true This file system is mounted by the **mount all** (or mount -a) command.

After a value of true or false, the following can be specified:

readonly The filesystem is mount read-only by default.

options See *-o* options in the **MOUNT** command in *AIX Operating System Commands Reference*.

site The name of the machine that contains the minidisk.

size Used by the **mkfs** command for reference and to build the file system. The value is the number of blocks in the file system.

skip Used by the **mkfs** command to initialize the free list and super block of a new file system. The value is the number of blocks to skip when the free list is interleaved. This number is processor- and device-specific.

type Used by the **mount** command to determine whether or not this file system should be mounted. When the command **mount -t string** is issued, all of the currently unmounted file systems with a **type** equal to **string** are mounted.

vol Used by the **mkfs** command when initializing the label on a new file system. The value is a volume or pack label using a maximum of eight characters. The file system label is always the stanza name.

quotas Used to mark which file system will be affected when a **-a** flag is specified with a **quotaon**, **quotaoff**, **quotacheck**, or

repquota command. If the value is "on", the file system is affected. Otherwise, it is ignored.

Example

```
    *
    * File system information
    *
default:
        vol       = "AIX"
        mount     = false
        check     = false
        free      = false
        backupdev =/dev/fd0
        backuplen = 1440

/u1:
        dev       = /dev/hd1
        vol       = "/u1"
        mount     = true
        check     = 1
        gfs       = 2
        gfspack   = 1
        size      = 12000
        ftype     = nonrepl
        free      = true
        site      = aixps

/aixps:
        dev       = /dev/hd2
        vol       = "/aixps"
        mount     = automatic
        check     = 0
        gfs       = 3
        gfspack   = 1
        size      = 6000
        ftype     = nonrepl
        free      = true
        site      = aixps

 /:
        dev       = /dev/hd3
        vol       = "/"
        mount     = automatic
        check     = 0
        gfs       = 1
        gfspack   = 1
        size      = 50000
        ftype     = repl,primary
        free      = true
        site      = aixps


*
/aixps/tmp:
        dev       = /dev/hd6
        vol       = "/aixps/tmp"
        mount     = true
        check     = 1
        gfs       = 4
```

```
        gfspack   = 1
        size      = 4000
        ftype     = nonrepl
        free      = true
        site      = aixps
*
/ul/aixps2:
        dev       = "aixps2:/ul"
        vol       = "/ul/aixps2"
        options   = "rw,intr,bg"
        gfs       = 101
        mount     = false
        checks    = false
        free      = true
        type      = nfs_mount
        ftype     = nfs
        site      = aixps
```

File
/etc/filesystems

Related Information
In this book: "attributes" in topic 2.3.5 and "fs" in topic 2.3.20.

The **backup**, **df**, **dumpbsd**, **fsck**, **mkfs**, **mount**, **restore**, and **umount** commands in *AIX Operating System Commands Reference*.

2.3.19 fonts

Purpose
Defines annotated and geometric character fonts (AIX PS/2 only).

Description
IBM supplies precompiled annotated text fonts with the AIX Operating System. For more information on how to use the fonts described in this section, see Chapter 6, "Advanced Display Graphics Support Library" in topic 2.6.

The system console provided by the **/dev/hft** device driver does not use fonts that can be modified by the user. The **display** command can be used to choose between either a hardware-generated font which displays code page P0 or a software-generated font which displays all the symbols in code pages P0, P1, and P2. See "display symbols" in topic 2.4.4.

The GSL-supported devices also recognize one geometric text font format that allows you to design your own set of characters. A geometric text font is also known as a programmable character set (PCS) font. The PCS font can be used on all GSL-supported devices.

Subtopics
2.3.19.1 Annotated Text Font Format
2.3.19.2 Geometric Text Font Format
2.3.19.3 rtfont File Format

2.3.19.1 Annotated Text Font Format

An annotated text font definition file has three major parts in the following sequence:

A header that describes the font. The header is the same for al annotated text fonts.

A set of condensed raster mosaics that describes each character in th font.

A look-up table that has an index entry to find each characte representation in the font.

Look-up table entries are 32 bits and each describes the start of it raster mosaics entry, its width, and the whitespace compressed from the top and bottom of its raster mosaics entry.

Subtopics
2.3.19.1.1 Annotated Text Font Header
2.3.19.1.2 Annotated Text Font Raster Mosaics
2.3.19.1.3 Annotated Text Font Look-up Table
2.3.19.1.4 Annotated Text Font Files

2.3.19.1.1 Annotated Text Font Header

The annotated text font header is a fixed-length structure common to all annotated text fonts for all displays. The information in header fields are shown in the following table.

Offset in Bytes	**Length in Bytes**	**Field**	**Description**
0x00	4	DDDFSIZE	The size in bytes of the area containing the font and the look-up table.
0x04	2	fntclass	A number that uniquely identifies the format of the look-up table that follows: 0x01 = not a 5081 font 0x02 = a 5081 font
0x06	2	fntid	The name an application uses to identify a font. This must be a value within the range of 0 to 1024.
0x08	4	fntstyle	Font style.
0x0C	4	fntattr	Identifies the attributes of the font. Possible values are: 0x0000 - no special values 0x0001 - bold version of this font 0x0002 - italic version of this font
0x10	4	fnttotch	The total number of characters in the font. This is used to determine whether a specified character code is valid for this font.
0x14	4	fnttblsz	Total number of words in the font table.
0x18	2	fntbasln	The scan line within a character box of the baseline for characters in this font (zero origin).
0x1A	2	fntcapln	The scan line within a character box of the caps line for characters in this font (zero origin).
0x1C	2	fntcolmn	Width of character box in pels.
0x1E	2	fntrows	Height of character box in pels.
0x20	2	fntchrbt	Total number of bits per character.

0x22	2	fntultop	The scan line within the character box of the top line in the underscore (zero origin).
0x24	2	fntulbot	The scan line within the character box of the bottom line in the underscore (zero origin).
0x26	1	fntmonpt	Mono pitch flag in leftmost bit of this byte.
0x28	4	fntlkup	Byte offset from the beginning of this structure to the beginning of the font look-up table.

2.3.19.1.2 Annotated Text Font Raster Mosaics

This contains a definition for each character in the font. Each character is entered in this area with the horizontal slices bit-packed one right after the other. The first bit of the first character slice is forced to begin in the most significant bit of a byte. The raster mosaics start immediately after the header (0x2C from the start address of the font structure). See *Annotated Text Example One* on 2.3.19.1.3.

2.3.19.1.3 Annotated Text Font Look-up Table

The look-up table immediately follows the raster mosaic. There is one 32-bit look-up table entry for each character in the font. The look-up table can be found by adding the value **fntlkup** given in the header to the starting address of the font structure. The table entry for any given character is found by using the font position number as an index into the table. (See "display symbols" in topic 2.4.4 for a list of the font position numbers.) Each look-up table entry contains the following fields:

Offset in Bits	Length in Bits	Field	Description
0	5	lkup_top	The number of blank scan lines that have been eliminated from the top of this character raster image (whitespace).
5	5	lkup_bot	The number of blank scan lines that have been eliminated from the bottom of this character raster image (whitespace).
10	6	lkup_widt	Contains the width in pels of this particular character.
16	16	lkup_ref	Byte offset from the start of the raster mosaics of the first scanline of the character's raster image.

Annotated Text Example One
See Figure 3-1 for this example. The character chosen is a capital A. This is shown as it would appear on the display and how it would be stored in the raster mosaics. Also shown is the font look-up table entry for this character. Note that the data associated with the top and bottom two scan lines of the character image do not appear in the raster mosaics since they consist of zeros.

To reconstruct the character image from the raster mosaics, it is necessary to use the font look-up table. The display symbol code associated with the character that is to be displayed is used to access its corresponding 4-byte entry in the font look-up table. The information contained in a font look-up table entry is shown. The capital Ts represent the bits containing the number of top blank scan lines that were compressed from the character image. The capital Bs represent the bits containing the number of bottom blank scan lines that were compressed from the character image. The capital Ws represent the bits containing the width in pels of this character. Capital Os represent the bits containing the offset of the compressed portion of this character image data in the raster mosaics. For this example, the value associated with T is 2, the value associated with B is 2, and the width (W) is 5. The value associated with O is the offset of the **y**(th) byte of the raster mosaics.

Figure 3-1. Example of Annotated Text Font Storage

If this font is defined in a file named **/usr/lpp/gsl/font_src/nrm1.9x20.s**, then compile it using the following commands:

```
cd      /usr/lpp/gsl
cc      font_src/nrm1.9x20.s  -o fonts/nrm1.9x20
```

2.3.19.1.4 Annotated Text Font Files

/usr/lpp/gsl/fonts/nrm1.4x8 Normal 4 by 8 micro font, compiled
/usr/lpp/gsl/fonts/nrm1.6x9 Normal 6 by 9 font, compiled
/usr/lpp/gsl/fonts/nrm1.6x11 Normal 6 by 11 font, compiled
/usr/lpp/gsl/fonts/nrm1.7x15 Normal 7 by 15 font, compiled
/usr/lpp/gsl/fonts/nrm1.7x22 Normal 7 by 22 font, compiled
/usr/lpp/gsl/fonts/nrm1.8x14 Normal 8 by 14 font, compiled
/usr/lpp/gsl/fonts/bld1.9x20 Bold 9 by 20 font, compiled
/usr/lpp/gsl/fonts/erg1.9x20 Ergonomic 9 by 20 font, compiled
/usr/lpp/gsl/fonts/itl1.9x20 Italic 9 by 20 font, compiled
/usr/lpp/gsl/fonts/nrm1.9x20 Normal 9 by 20 font, compiled
/usr/lpp/gsl/fonts/bld1.11x23 Bold 11 by 23 font, compiled
/usr/lpp/gsl/fonts/nrm1.11x23 Normal 11 by 23 font, compiled
/usr/lpp/gsl/fonts/nrm1.12x30 Normal 12 by 30 font, compiled
/usr/lpp/gsl/fonts/nrm1.18x40 Normal 18 by 40 title font, compiled

Many of the fonts are also supplied in rotated versions (for instance, nrm1.9x20.90 is the 90° rotated version of nrm1.9x20).

2.3.19.2 Geometric Text Font Format

Geometric text fonts are also known as programmable character set (PCS) fonts and they can be used on all GSL supported devices. Each character is defined as a series of moves or draws that define the shape of the character. The moves and draws are specified as X-Y pairs of signed relative values (relative to the previous ending point, or to the bottom left of the character box for the first X-Y pair). The range of the incremental values for the X and Y coordinates is -64 to +63.

Each character definition in the font consists of a 2-byte length field for the character definition followed by 2-byte X-Y entries:

```
Length of definition          2 bytes
sXXXXXX 1  sYYYYYY b          2 bytes
sXXXXXX 1  sYYYYYY b          2 bytes
sXXXXXX 1  sYYYYYY b          2 bytes
sXXXXXX 1  sYYYYYY b          2 bytes
  .     .      .   .
  .     .      .   .
  .     .      .   .
  .     .      .   .
sXXXXXX 1  sYYYYYY b          2 bytes
```

s is the sign bit (0 = positive, 1 = negative). Negative values are in twos complement notation.

b is the blanking bit. If b = 1, the primitive is blanked causing movement without display.

1 is the low order bit of the X coordinate field and must always be a 1.

If the first X-Y pair is a draw rather than a move, the line is drawn from the bottom left corner of the character box. A move is specified by the low-order bit of the Y coordinate being on. A draw is specified by the low-order bit being off. The last X-Y pair in the series for the character is defined by the length field.

Subtopics
2.3.19.2.1 Geometric Text Font Definition File

2.3.19.2.1 Geometric Text Font Definition File

The PCS font definition file consists of:

A header that contains identifier and control informatio
A table of index values used to find each character definitio
The character definitions

Offset in Bytes	**Length in Bytes**	**Field**	**Description**
0x00	2	length	The length of the PCS descriptor record inc length field.
0x02	4	Reserved	0x00000000
0x06	1		Bit 1 = 0 - EBCDIC = 1 - ASCII Bits 1-2 = Reserved Bits 3-7 = (Type) specifies the data form definition for programmable characters. One is defined: '00001'B = Type 1
0x07	1	Reserved	Must be 0.
0x08	2	fontid	This field identifies the programmable char Font IDs within the range of 1025 to 3267 a reserved for 1-byte character sets. IDs wi range of 32768 to 65535 are reserved for 2- character sets.
0x0A	1	segmented	For 2-byte character sets, this byte contai first byte of the 2-byte character code.
0x0B	1	Reserved	Must be 0.
0x0C	2	P	Range of X (between 0 and P)
0x0E	2	Q	Range of Y (between 0 and Q)
0x10	1	CP0	Starting character code within PCS (within of 0x21 to 0xFE.)
0x11	1	CP**n**	The last character code within this PCS. I 0, 0xFE is assumed. CP**n** must not be less t
0x12	2	font baseline	The value of the font baseline in pixels ir direction from the bottom line of the chara This value is used in conjunction with the alignment function.
0x14	2	font capline	The value of the font capline in pixels in direction from the bottom line of the chara This value is used in conjunction with the alignment function.

0x16	1	Reserved	
0x17	1	Default error code point	This is the character code within this PCS specifies the character to be displayed whe invalid code is encountered or the characte does not exist.
0x18	var	Character Index	This field contains 2-byte offsets to each description. Each offset is from the begir the descriptor record.
var	var	Character Descripti	This field contains the character definitic beginning with code point CP0, in ascending

P and Q together define the character box within which a normal character will fit. The values of P and Q are defined in device coordinate space (pixels) and control spacing between characters and new line spacing. The bottom left corner of the box is 0,0 and the top right corner is P,Q. Characters can extend outside this box as P and Q control only the intercharacter spacing. You can override the value of P specified in the header by specifying a character inline spacing value greater than 0. Undefined character codes (outside the range CP0-CP**n**, or those with an index value of 0) are displayed as the default code point character.

Each character index value is the offset from the start of the header record to the actual character definition. The index must always be represented in its entirety, even if not all of the characters in the code range are defined. For example, the maximum length of the index, if CP0 is specified as 0x41 and CP**n** as 0xFF, is 191 multiplied by 2 bytes. For undefined characters, the index value should contain an offset to the default code definition.

Each character definition begins with a 2-byte length field which specifies the length of the character definition including the length field.

2.3.19.3 rtfont File Format

The **rtfont** format is provided for use with X-Windows, Version 1.1 or higher, and with the Advanced Display Graphics Support Library, Version 2.2 or higher. An **rtfont** definition file contains three major data structures in the following order:

1. A header structure, called **FONT_HEAD**, which contains information about the font in general. The header also contains offsets to the following two structures.

2. A character index array structure, called **CHAR_INDEX**, which contains modifiers of subsequent indices in the form of character offsets into the character data structure, also known as the character ***glyphs***. The high-order four bits of each character index entry contain control information that defines how to interpret the 28 low-order bits of each entry.

3. A character glyph structure, called **CHAR_GLYPH**, which contains the character image data and information unique to each character.

Subtopics

2.3.19.3.1 Header Structure for rtfont Format

The header is a fixed-length structure common to all fonts in the rtfont format. This structure, called **FONT_HEAD**, is defined in the **rtfont.h** header file, and it contains the following members:

Offset in Bytes	**Length in Bytes**	**Field**	**Description**
0x00	16	font_id	The ID of the font.
0x10	4	off_to_index	The offset in bytes from the beginning of the file to the character index array.
0x14	4	off_to_glyphs	The offset in bytes from the beginning of the file to the character glyphs.
0x18	4	glyph_format	The format of the character image data.
0x1C	4	src_file_type	The type of font file from which this file was generated.
0x20	4	design_size	The design size in (1/2)**16 points. This field is for use by T[E]X, a typesetting system described in *The T[E]Xbook*, Donald E. Knuth, Addison Wesley Publishing Company, 1986. (See Note 1.)
0x24	4	check_sum	The checksum (T[E]X). (See Note 1.)
0x28	4	hppp	The number of horizontal pixels per point x 2**16. (See Note 1.)
0x2C	4	vppp	The number of vertical pixels per point x 2**16. (See Note 1.)
0x30	4	fiFlags	The flags field bits. (See Note 1.)
0x34	4	firstCol	Reserved (set to 0). (See Note 1.)
0x38	4	lastCol	Reserved (set to 0). (See Note 1.)
0x3C	4	firstRow	Reserved (set to 0). (See Note 1.)

0x40	4	lastRow	Reserved (set to 0). (See Note 1.)
0x44	4	nProps	The number of properties. (See Note 1.)
0x48	4	chDefault	The default character. (See Note 1.)
0x4C	4	font_Descent	Extent below baseline for spacing. (See Note 1.)
0x50	4	font_Ascent	The extent above baseline for spacing. (See Note 1.)
0x54	20	minbounds	The minimum glyph metrics over all characters in font.
0x68	20	maxbounds	The maximum glyph metrics over all characters in font.
0x7C	4	pixDepth	The number of intensity bits per pixel. (See Note 1.)
0x80	4	glyphSets	The number of sets of glyphs. (See Note 1.)
0x84	4	raster_align	The scanline alignment for glyphs.
0x88	4	index_width	The width of each entry in the character index array.
0x8C	4	char_width	Fixed width of each character. (If set to 0, characters are variable width.)
0x90	4	char_height	Fixed height of each character. (If set to 0, characters are variable height.)
0x94	4	last_index	The last character index in the font.
0x98	4	codepoints_less_1	The number of code points minus 1.
0x9C	16	app_var	For application use.
0xAC	64	pad	Reserved (set to 0).

Notes:

1. This value is not supported by the Advanced Display Graphics Support Library or by the X-Windows Version 1.1 program.

2. This value is not supported by the X-Windows Version 1.1 program.

Some of the fields in the header structure require special values, as explained in the following list.

glyph_format This field has two possible values:

RASTER_DATA All glyphs are in raster format bit image.

PK_DATA All glyphs are in PK format. (See Note 1.)

src_file_type This field has one of the following values:

VRM_SRC_FILE The origin of this font file was a VRM fonts file, provided with the AIX Operating System.

PK_SRC_FILE The origin of this font file was a PK font file, a font structure generated by the METAFONT compiler. METAFONT is described in *The METAFONT Book*, Donald E. Knuth, Addison Wesley Publishing Company, 1986. For details on developing various font styles and sizes with METAFONT, refer to *Computer Modern Typefaces*, Donald E. Knuth, Addison Wesley Publishing Company, 1986.

X10_SRC_FILE The origin of this font file was an X10 font file, distributed with the Massachusetts Institute of Technology X-Windows program, Version 10.4.

X11_SRC_FILE The origin of this font file was an X11 font file, distributed with the Massachusetts Institute of Technology X-Windows program, Version 11.

ORIG_SRC_FILE This is the original source file for this font.

raster_align Raster format glyphs are stored beginning on a 32-bit boundary with scanlines packed from end to beginning. The first scanline of a raster format glyph is always on a 32-bit boundary. The significant scanline bits begin at the boundary of the scanline and continue for the width of the character. The boundary for subsequent scanlines is defined by the **raster_align** field as follows:

NO_ALIGN The first scanline is on a 32-bit boundary and subsequent scanlines for this character are bit packed. That is, the second scanline begins in the bit following the last bit of the first scanline, the third scanline follows the last bit of the second, and so on. (See Note 1.)

BYTE_ALIGN The first scanline is on a 32-bit boundary and subsequent scanlines for this character begin on the next 8-bit boundary following the last bit of the previous scanline. (See Note 1.)

HWD_ALIGN The first scanline is on a 32-bit boundary and

subsequent scanlines for this character begin on the next 16-bit boundary following the last bit of the previous scanline.

FWD_ALIGN The first scanline is on a 32-bit boundary and subsequent scanlines for this character begin on the next 32-bit boundary following the last bit of the previous scanline. (See Note 2.)

minbounds, maxbounds

These fields contain the maximum and minimum values of each individual **BOUNDS** structure for **all** characters in the font, as explained in the following section on the **CHAR_GLYPH** structure. The **minbounds.rbearing** value, for instance, must exist in at least one individual **BOUNDS** structure and the **rbearing** field in any other individual **BOUNDS** structure must be greater than or equal to this value.

2.3.19.3.2 Character Index Array for rtfont Format

Since the data bytes within the data stream are used to access the character index array, the array must contain at least 256 entries. Entries for which a character is not defined should be set to the offset values of a valid default character. One such valid offset is zero. Since each font has at least one character defined, there is always a **first** character pointed to by offset zero, the first character in the glyph structure.

The character index information is contained in the **CHAR_INDEX** array, as defined in the **rtfont.h** header file. This structure contains the following elements:

Offset in Bytes	**Length in Bytes**	**Field**	**Description**
0x00	4	CHAR_INDEX	Character glyph offset or index modifier.

The low-order 28 bits of a character index array can be either an offset into the character glyph structure or a modifier value that modifies the next offset into the character index array. The high-order four bits of each entry define how the low-order 28 bits are interpreted. The value of the **CHAR_INDEX** field, when logically ANDed with the following values, is used to interpret the low-order 28 bits.

INDBASE Specifies a base modifier, such as ANSI SG0, SG1, and so on. (See Note 1 in topic 2.3.19.3.1.)
INDMOD Specifies an index modifier, such as ANSI SS1, SS2, and so on.
INDCPT Indicates that the data stream source byte is not a code point.
INDMASK Specifies an offset to a glyph whose data stream source byte is a code point.

If the data source byte is not a code point and the low-order 28 bits do not indicate a base or index modifier, those bits **must** be a valid offset into the glyph structure.

2.3.19.3.3 Character Index Example

A sample data stream processing algorithm follows:

```
  unsigned base_modifier, index_modifier, value, n;
     Both base_modifier and index_modifier are to be initialized
     to 0 for each independent character stream to be processed;

  if  ( (n=data_stream+index_modifier+base_modifier)<=FONT_HEAD.last_index)
  {
     index_modifier=0;
     value=CHAR_INDEX[n]
     if  (value & INDBASEU)
     {
        base_modifier=value & !INDMASK
        process next character in data stream
     }
     else if  (value & INDMOD)
     {
        index_modifier=value & !INDMASK
        process next character in data stream
     }
     else
     {
        glyph_offset=value & !INDMASK
        process glyph
        process next character in data stream
     }
  }
  else
     offset into CHAR_INDEX is in error
```

2.3.19.3.4 Character Glyph Structure for rtfont Format

The character glyph structure contains information pertinent to each character in the font. The information per character is defined via the **CHAR_GLYPH** structure, as defined in the **rtfont.h** header file. This structure contains the following members:

Offset in Bytes	**Length in Bytes**	**Field**	**Description**
0x00	1	pk_format_flag	METAFONT information. (See Note 1 in topic 2.3.19.3.1.)
0x01	3	resv	Reserved field (set to 0.)
0x04	4	tfm	T[E]X font metric information. (See Note 1 in topic 2.3.19.3.1.)
0x08	20	char_bounds	Character **BOUNDS** structure, defined below.
0x1C	var	char_bits	Character image.

2.3.19.3.5 Bounds Structure for rtfont Format

The **BOUNDS** structure, referenced by the **CHAR_GLYPH** structure above, is defined in the **rtfont.h** header file, and it contains the following members:

Offset in Bytes	Length in Bytes	Field	Description
0x00	4	rbearing	Character origin to right edge of raster. (See Note 2 in topic 2.3.19.3.1.)
0x04	4	lbearing	Character origin to left edge of raster. (See Note 2 in topic 2.3.19.3.1.)
0x08	4	descent	Baseline to bottom edge of raster. (See Note 2 in topic 2.3.19.3.1.)
0x0C	4	ascent	Baseline to top edge of raster. (See Note 2 in topic 2.3.19.3.1.)
0x10	4	width	Advance to next character origin.

The glyph, or character data, can be drawn relative to any point in a given x,y coordinate system. The following description of the **BOUNDS** variables assumes the x,y position is on the baseline of the character. Coordinates are positive to the right and positive in the downward direction. The ***pel box*** is the area where the glyph is positioned on the screen when the **rtfont** is used.

The following diagram graphically portrays each of the fields in the **BOUNDS** structure and shows how these fields define the pel box, relative to the coordinates x and y. In this example:

1. The left vertical edge of the pel box is located at **x + lbearing**.
2. The right vertical edge of the pel box is located at **x + rbearing**.
3. The upper horizontal edge of the pel box is located at **y - ascent**.
4. The lower horizontal edge of the pel box is located at **y + descent**.
5. The origin for the next character is at the point **(x + width, y)**.
6. The width of the pel box, which defines the number of scan columns, is **rbearing - lbearing + 1**.
7. The height of the pel box, which defines the number of scan lines, is **ascent + descent + 1**.

Figure 3-2. Example of an **rtfont** Pel Box

(x,y) is the position from which this character's pel box is referenced. (x1,y1) is the position for the next character's pel box reference point.

Files

/usr/include/rtfont.h	Header file for the rtfont format.
/usr/bin/vrm2rtfont	Font conversion command.

Related Information
In this book: "master" in topic 2.3.32, "data stream" in topic 2.4.3, "display symbols" in topic 2.4.4, "gsgtat" in topic 2.6.28, "gsgtxt" in topic 2.6.29, "gstatt" in topic 2.6.52, "gstext" in topic 2.6.54, "gsxtat" in topic 2.6.60, and "gsxtxt" in topic 2.6.61.

The **display** command in *AIX Operating System Commands Reference*.

2.3.20 fs

Purpose
Contains the format of a file system volume.

Synopsis

```
#include <sys/types.h>
#include <sys/filsys.h>
```

Description
Every file system storage volume has a common format for certain vital information. Every such volume is divided into a certain number of 4096-byte sectors. Sector 0 is unused and is available to contain a bootstrap program, pack label, or other information.

Sector 1 is the super block. The super block describes the layout of the file system. The sectors immediately following the super block contain the inodes (see "inode" in topic 2.3.29). Each allocated inode describes a file in the file system. The remaining sectors of the file system are a pool of blocks from which the data blocks of files are allocated.

A file system volume may be designated as being one of several replicated copies of a file system. When this is done, each copy (also called a pack) of the file system is given the same global file system number, and each copy, when mounted, must be mounted on separate sites and at the same place in the network-wide directory hierarchy (see "mount" in topic 1.2.172). All copies of a replicated file system must agree on the read-only status with which they are mounted.

Each copy of a replicated file system must have the same number of inodes, thus allowing each copy to have the potential of storing any of the files in the file system. However, each copy of the file system need not actually store the data for all of the files in the file system. Because of this, the copies of a file system need not be the same size. Certain copies may have a smaller pool of blocks from which to allocate data blocks for files.

One copy of a replicated file system is designated the primary copy. This copy must be mounted on one of the sites in the network (called the primary site for this file system) in order for any user or process to make a change to any file in the file system. When a change is made to a file, it is made first to the primary copy, and after the change has been committed (see "fsync, fcommit" in topic 1.2.87), the other sites which store this file are updated. If a copy of the file system is not mounted, is on a site which is down, or is on a site which is not in communication with the primary site, the propagation of the changed file to this copy is done automatically at a later time. If none of the currently mounted copies of a replicated file system is the primary copy, the files in the file system are treated as read-only.

A copy other than the primary copy of a file system may be designated as storing all of the files in the file system. Such a copy is called a backbone copy of the file system, and the site on which it is mounted is called a backbone site.

The remaining copies of the file system, those not designated as the primary copy or as a backbone copy, selectively store files according to the **fstore** attribute of each file (see "chfstore" in topic 1.2.41). The interpretation of a file's **fstore** attribute is different for file systems

designated as system-replicated than for file systems designated as user-replicated.

In a system-replicated file system, the data pages of a file are stored in a non-primary, non-backbone copy if the logical AND of the file's **fstore** value and the super block's **fstore** value is nonzero. The program **chfstore** or the system call **chfstore** is used to change a file's **fstore** value if the file is in a system-replicated file system.

In a user-replicated file system, the data pages of a file are stored in a nonprimary, non-backbone copy which has pack number **packno** if the **packno** bit of the file's **fstore** value is set. The low order bit (0x1) of the **fstore** value represents pack number 1, while the high order bit (0x80000000) represents pack number 32. The program **store** or the system call **chfstore** is used to change a file's **fstore** value if the file is in a user-replicated file system.

The layout of the super block as defined by the include file **<sys/filsys.h>** is:

```
  #define NICFREE    600
  #define NICINOD    325
  #define NCMTLST    200
  #define NGENLST    25

  /* number of freeblock pointers in the super block for replicated file syste
  #define NICSFREE   NICFREE \
                     - (NGENLST * sizeof(fsgen_t) \
                     + NCMTLST * sizeof(commitcnt_t)) / sizeof(daddr_t)


  /*
   * Structure of the super block
   */
  struct  filsys
  {
      long           s_magic;     /* identifies this as a TCF file system */
                                  /* defined as a constant below */
      gfs_t          s_gfs;       /* global file system number */
      daddr_t        s_fsize;     /* size in blocks of entire volume */
      commitcnt_t    s_lwm;       /* all prior commits propagated */
      commitcnt_t    s_hwm;       /* highest commit propagated */
      /* oldest committed version in the list.
       * llst mod NCMTLST is the offset of commit #llst in the list,
       * which wraps around from there.
       */
      commitcnt_t    s_llst;
      fstore_t       s_fstore;    /* file system storage bit mask; if the
                                     filsys is replicated and this is not a
                                     primary or backbone copy, this bit mask
                                     determines which files are stored */

      time_t         s_time;      /* last super block update */
      daddr_t        s_tfree;     /* total free blocks*/

      ino_t          s_isize;     /* size in blocks of i-list */
      short          s_nfree;     /* number of addresses in s_free */
      unsigned short s_flags;     /* filsys flags, defined below */

  #define s_ronly s_flags
```

```
    ino_t          s_tinode;    /* total free inodes */
    ino_t          s_lasti;     /* start place for circular search */
    ino_t          s_nbehind;   /* est # free inodes before s_lasti */
    pckno_t        s_gfspack;   /* global file system pack number */
    short          s_ninode;    /* number of inodes in s_inode */
    short          s_dinfo[4];  /* interleave stuff */
#define s_m     s_dinfo[0]
#define s_skip  s_dinfo[0]
#define s_n     s_dinfo[1]
#define s_cyl   s_dinfo[1]
    char           s_flock;     /* lock during free list manipulation */
    char           s_ilock;     /* lock during i-list manipulation */
    char           s_fmod;      /* super block modified flag */
    char           s_version;   /* version of the data format in fs. */
    char           s_fsmnt[32]; /* name of this file system */
    char           s_fpack[8];  /*name of this physical vol.*/

#define s_fname s_fsmnt                /* for backwards compatibility */
    ino_t          s_inode[NICINOD];   /* free inode list */
    union {   /* union of replicated and non-replicated filsystem types */
        daddr_t su_free[NICFREE];      /* free block list
                                          for non-replicated filsys */
        struct {
            daddr_t     su_sfree[NICSFREE]; /* free block list
                                               for replicated filsys */
            ino_t       su_cmtlst[NCMTLST]; /* list of recent*/
                                            /* commits         */
            commitcnt_t su_mntcnt;          /* prim site mounts */
            fsgen_t     su_fsgens[NGENLST]; /* filsys generations */
        } su_st;
    } s_un;
#define s_free     s_un.su_free
#define s_cmtlst   s_un.su_st.su_cmtlst
#define s_fsgens   s_un.su_st.su_fsgens
#define s_mntcnt   s_un.su_st.su_mntcnt
    char           s_byteorder;       /* byte order of integers */

};

        /* Current magic number */
#define SB_MAGIC        0xffeeddcd    /* word unlikely to be found in non-fs
```

s_magic contains the value SB_MAGIC and identifies this storage medium as a valid AIX file system.

s_version is a version number for the format of the file system. It is changed whenever the structure of the super block or inodes change in an incompatible way.

The global file system number, **s_gfs**, is the unique name for this file system. Each file system has a distinct **s_gfs** number. Each copy of a replicated file system has the same **s_gfs** number, but has a distinct **s_gfspack** number. To uniquely identify a file in AIX, the pair *gfs,inumber* is used. To uniquely identify a copy of a replicated file, the triple *gfs, gfspack, inumber* is used.

s_isize is the address of the first block after the i-list, which starts just after the super block, block 2. Thus the i-list is **s_isize**-2 blocks long. **s_fsize** is the address of the first block not potentially available for allocation to a file. These numbers are used by the system to check

for bad block addresses; if an "impossible" block address is allocated from the free list or is freed, a diagnostic message is written on the on-line console. Moreover, the free array is cleared, so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The **s_free** array contains, in **s_free [1]**, ... , **s_free [s_nfree-1]**, up to NICFREE free block numbers. **s_free [0]** is the block address of the head of a chain of blocks constituting the free list. In a replicated file system, the number of free block numbers is only NICSFREE.

The layout of each block of the free chain as defined in the include file **<sys/fblk.h>** is:

```
struct fblk
{
    long  df_nfree;
    daddr_t df_free[NICFREE];
};
```

The fields **df_nfree** and **df_free** in a free block are used exactly like **s_nfree** and **s_free** in the super block. To allocate a block: decrement **s_nfree**, and the new block number is **s_free [s_nfree]**. If the new block address is 0, there are no blocks left, so give an error. If **s_nfree** became 0, read the new block into **s_nfree** and **s_free**. To free a block, check if **s_nfree** is NICFREE; if so, copy **s_nfree** and the **s_free** array into it, write it out, and set **s_nfree** to 0. In any event set **s_free [s_nfree]** to the freed block's address and increment **s_nfree**.

s_cmtlst is a list of the last NCMTLST changed files. **s_llst** (module NCMTLST) is the index of the oldest entry in the list. **s_hwm** (module NCMTLST) is the index of the newest entry in the list; that is, the high water mark of commits seen. If **s_lwm** is within NCMTLST of **s_hwm**, **s_lwm** (module NCMTLST) is the index of the most recent entry in the list up to which every entry has been processed and the local file brought up to date; that is, the low water mark of commits propagated. These three indices always follow the relationship that **s_llst <= s_lwm <= s_hwm**. In the primary copy of the file system, **s_lwm** and **s_hwm** are always equal.

s_ninode is the number of free i-numbers in the **s_inode** array. To allocate an inode: if **s_ninode** is greater than 0, decrement it and return **s_inode [s_ninode]**. If it was 0, read the i-list and place the numbers of all free inodes (up to NICINOD) into the **s_inode** array, then try again. To free an inode, provided **s_ninode** is less than NICINODE, place its number into **s_inode [s_ninode]** and increment **s_ninode**. If **s_ninode** is already NICINODE, don't bother to enter the freed inode into any table. This list of inodes is only to speed up the allocation process; the information as to whether the inode is really free or not is maintained in the inode itself.

The fields **s_lasti** and **s_nbehind** are used to avoid searching the inode list from the beginning each time the system runs out of inodes. **s_lasti** gives the base of the block of inodes last searched on the file system when inodes ran out, and **s_nbehind** gives the number of inodes, whose numbers were less than **s_lasti** when they were freed with **s_ninode** already NICINODE. Thus **s_nbehind** is the number of free inodes before **s_lasti**. The system will search forward for free inodes from **s_lasti** for more inodes unless **s_nbehind** is sufficiently large, in which case it will search the file system inode list from the beginning. This mechanism serves to avoid **n****2 behavior in allocating inodes.

s_flock and **s_ilock** are flags maintained in main memory while the file system is mounted and their values on disk are immaterial. The value of **s_fmod** on disk is likewise immaterial; it is used as a flag to indicate that the super block has changed and should be copied to the disk during the next periodic update of file system information.

s_flags is a set of flag bits whose values are defined as follows:

```
/* Flags used in s_flags */
#define SB_RDONLY   0x1  /* no writes allowed on file system         */
#define SB_CLEAN    0x2  /* file system was unmounted cleanly        */
#define SB_DIRTY    0x4  /* file system was mounted without clean bit set *
#define SB_PRIMPACK 0x10 /* This is primary pack of the file system  */
#define SB_REPLTYPE 0x20 /* This is a replicated type of file system */
#define SB_USER     0x40 /* This is a user-replicated file system    */
#define SB_BACKBONE 0x80 /* This is a backbone copy of file system   */
```

The **SB_RDONLY** flag is used only in main memory to indicate that the file system is mounted read only. The **SB_CLEAN** flag is used to indicate that a file system has been unmounted cleanly or that file system checks have been successfully run. This indicates that file system checks are not necessary when the file system is next mounted. The **SB_DIRTY** flag is to guard against the **SB_CLEAN** flag being turned on when an unchecked file system is mounted and then unmounted.

The **SB_REPLTYPE** flag indicates that this is a replicated file system. If it is a replicated file system, the **SB_PRIMPACK** flag identifies the primary copy -- the only copy of the replicated file system that can be modified. The **SB_BACKBONE** flag indicates a site which will store a complete copy of the file system. **SB_PRIMPACK** and **SB_BACKBONE** should not both be set. The **SB_USER** flag identifies a replicated file system as being a user-replicated file system, as opposed to a system-replicated file system. All copies of a file system should agree on whether the file system is user-replicated or system-replicated. As described above, the system interprets the file storage value (**fstore**) of a file in a user-replicated and system-replicated file system differently. In user-replicated file systems, the **fstore** bits refer to file system pack numbers; while in system-replicated file systems, the **fstore** bits of a file are compared to the **s_fstore** field in the super block. The **s_fstore** field is only relevant if neither **SB_BACKBONE** nor **SB_PRIMPACK** is set. System-replicated and user-replicated file systems also differ in the way symbolic links are deleted. The system call **rmslink** must be used to remove a symbolic link from a system-replicated file system. This restriction is to prevent symbolic links in the replicated root file system from being deleted inadvertently.

s_time is the last time the super block of the file system was changed.

The total number of unallocated blocks in a file system is maintained in the field **s_tfree**. In a replicated file system, this number may be different in the individual copies of the file system. The total number of unallocated inodes in a file system is maintained in the field **s_tinode**. In a replicated file system, this number is only maintained at the primary site. The value of **s_tinode** stored in the super block of other copies is undefined.

The fields **s_fpack** and **s_byteorder** are not currently maintained.

Files

/usr/include/sys/filsys.h
/usr/include/sys/stat.h

Related Information
In this book: "chfstore" in topic 1.2.41 and "inode" in topic 2.3.29.

The **chfstore** and **store** commands in *AIX Operating System Commands Reference*.

2.3.21 fsmap

Purpose
Centralizes file system characteristics for all nodes in a network.

Description
This file contains a superset of the attributes in the **/etc/filesystems** file.

Subtopics
2.3.21.1 Network File System Attributes

2.3.21.1 Network File System Attributes

Each stanza names the directory where the file system is normally mounted. The attributes specify all of the parameters of the file system. See "attributes" in topic 2.3.5 for the format of an attribute file. The attributes used are the same as those specified in **/etc/filesystems**.

Example

```
     *
     * File system information
     *
  default:
          vol       = "AIX"
          mount     = false
          check     = false
          free      = false
          backupdev = /dev/rfd0
          backuplen = 1440
  *
   /:
          dev       = /dev/hd02
          vol       = "/"
          mount     = automatic
          check     = 0
          gfs       = 1
          gfspack   = 1
          type      = repl,primary
          free      = true
          site      = aix
  *
  /aix:
          dev       = /dev/hd03
          vol       = "/aix"
          mount     = automatic
          check     = 0
          gfs       = 2
          gfspack   = 1
          type      = nonrepl
          free      = true
          site      = aix


  *
  /u:
          dev       = /dev/hd01
          vol       = "/u"
          mount     = true
          check     = 1
          gfs       = 3
          gfspack   = 1
          size      = 6000
          type      = nonrepl
          free      = true
          site      = aix
```

File

/etc/fsmap

Related Information

In this book: "attributes" in topic 2.3.5 and "filesystems" in

topic 2.3.18.

The **backup**, **df**, **fsck**, **mkfs**, **mount**, **restore**, and **umount** commands in *AIX Operating System Commands Reference*.

2.3.22 fspec

Purpose
Specifies formatting within text files.

Description
A text file format specification normally occurs in the first line of a text file. This format specifies how tabs expand in the remainder of the file.

A format specification consists of a sequence of parameters separated by blanks and enclosed by the brackets **<:** and **:>**. Each parameter consists of a key-letter, possibly followed immediately by a value. The following parameters are recognized:

d The **d** parameter takes no additional value. It indicates that the line containing the format specification is to be deleted from the converted file.

e The **e** parameter takes no additional value. It indicates that the current format prevails until another format specification is encountered in the file.

mmargin The **m** parameter specifies a number of spaces added to the beginning of each line. The value of **margin** must be an integer.

ssize The **s** parameter specifies a maximum line size. The value of **size** must be an integer. Size checking is performed after tabs are expanded, but before inserting the margin.

ttabs The **t** parameter specifies the tab settings for the file. The value of **tabs** must be one of the following:

A list of column numbers separated by commas, indicating tabs set at the specified columns.

A **-** (dash) followed immediately by an integer **n**, indicating tabs at intervals of **n** columns.

A **-** (dash) followed by the name of a supplied tab specification.

Standard tabs are specified by **t-8**, or the equivalent **t1, 9, 17, 25,** and so on. The **tabs** command defines the supplied tabs.

Default values assumed for parameters not supplied are **t-8** and **m0**. If the **s** parameter is not specified, no size checking is performed. If the first line of a file contains no format specification, the previous defaults are assumed for the entire file.

The format specification can be entered as a comment. In that case it is not necessary to code the **d** parameter.

Example
The following is an example of a line containing a format specification:

```
* <:t5,10,15 s72:> *
```

Related Information

The **ed, newform,** and **tabs** commands in *AIX Operating System Commands Reference*.

2.3.23 fstore

Purpose
Maps **fstore** bit patterns to CPU types.

Description
The **fstore** file explicitly maps **fstore** bit patterns to CPU types. This permits a convenient naming convention for controlling the replication of files in a system-replicated file system such as the root file system using the **chfstore** command.

The **fstore** file is an ASCII file composed of entries that are position-dependent and have the following format:

pattern:0:CPU-type

Each entry is delimited by a new-line. There are no limits imposed on the number of entries within the **fstore** file. The second field is reserved for future use. The remaining fields of an entry are:

pattern
This is an octal integer representing the **fstore** bit pattern.

0 Reserved.

CPU-type
This is the ASCII string for the CPU type corresponding to this **fstore** bit pattern. This string matches the CPU type string found in **/etc/site**.

This file is used by the **where** command to make inferences about file systems which are not currently mounted anywhere in the cluster.

File
/etc/fstore

Related Information
In this book: "chfstore" in topic 1.2.41.

The **chfstore** and **where** commands in the *AIX Operating System Commands Reference*.

2.3.24 gettydefs

Purpose
Describes speed and terminal settings used by the **getty** command.

Description
The **/etc/gettydefs** file contains information used by the **getty** command to set up the speed and terminal settings for a line. Since **/etc/gettydefs** is a symbolic link to **<LOCAL>/gettydefs**, the information contained therein is specific to the individual cluster site. The **/etc/gettydefs** file supplies information on what the login prompt should look like. It also supplies the speed to try next if the user indicates that the current speed is not correct by typing a *<break>* character.

Each entry in **/etc/gettydefs** has the following format:

label# initial-flags # final-flags #login-prompt #next-label

Each entry is followed by a blank line. Used within a line, the pound sign (#) acts as a field separator. Lines that begin with # are ignored and may be used to comment the file. The various fields can contain quoted characters of the form **\b, \n, \c,** and so on, as well as **\nnn**, where **nnn** is the octal value of the desired character. The various fields are:

label
This is the string against which **getty** tries to match its second argument. It is often the speed, such as **1200**, at which the terminal is supposed to run, but it need not be (see the paragraph at the end of this list).

initial-flags
These flags are the initial **ioctl** system call settings to which the terminal is to be set if a terminal type is not specified in the **getty** command. The **getty** command understands the symbolic names (see "termio" in topic 2.5.28). Normally, only the speed flag is required in the *initial-flags*. The **getty** command automatically sets the terminal to raw input mode and takes care of most of the other flags. The *initial-flag* settings remain in effect until **getty** executes login.

final-flags
These flags take the same values as the *initial-flags* and are set just before **getty** executes login. The speed flag is again required. The composite flag **SANE** takes care of most of the other flags that need to be set so that the processor and terminal are communicating in a rational fashion. The other two commonly specified *final-flags* are:

TAB3 causes tabs to be sent to the terminal as spaces.
HUPCLcauses the line to be hung up on the final close.

login-prompt
This entire field is printed as the *login-prompt*. Unlike the previous fields where blanks are ignored (a space, tab, or new-line), they are included in the *login-prompt* field.

next-label
This indicates the next *label* of the entry in the table that **getty** should use if the user types a *<break>* or the input cannot be read. Usually, a series of speeds are linked together in this fashion, into a closed set. For instance, **2400** linked to **1200**, which in turn is linked

to **300**, which finally is linked to **2400**.

If **getty** is called without a second argument, the first entry of **/etc/gettydefs** is used, thus making the first entry of **/etc/gettydefs** the default entry. This first entry is also used if **getty** cannot find the specified *label*. If **/etc/gettydefs** itself is missing, there is one entry built into the command which will bring up a terminal at **300** baud.

It is strongly recommended that after making or modifying **/etc/gettydefs**, it be run through **getty** with the check option to be sure there are no errors.

Files

/etc/gettydefs A symbolic link to **<LOCAL>/gettydefs**

<LOCAL>/gettydefs
The actual **gettydefs** file

Related Information
In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137 and "termio" in topic 2.5.28.

The **getty** and **login** commands in the *AIX Operating System Commands Reference*.

2.3.25 gps

Purpose
Used as the format for storing graphics file data as ***graphic primitive strings***.

Description
A graphic primitive string (***GPS***) is used to store graphical data in a particular format. The **plot** and **vtoc** commands produce GPS output files. Several commands edit and display GPS files on various devices. A GPS file is composed of as many as five types of graphical data or primitives:

comment A **comment** primitive is an integer string included within a GPS file that does not cause anything to be displayed. All GPS files begin with a **comment** of zero length.

lines A **lines** primitive has a variable number of points from which zero or more connected line segments are produced. The first point given produces a ***move*** to that location, relocating the graphics cursor without drawing. Successive points produce line segments from the previous point.

arc An **arc** primitive has a variable number of points to which a curve is fit. The first point produces a move to that point. If only two points are given, a line connecting the points is the result. If three points are given, a circular arc through the points is drawn. If more than three points are given, splines are fitted to connect the points.

text The **text** primitive draws characters beginning at a given point, with the first character centered on that point.

hardware The **hardware** primitive draws hardware characters or gives control commands to a hardware device. A single point locates the beginning location of the **hardware** string.

Graphic primitive strings are given as 16-bit units called ***command words***. The first command word determines the primitive type and sets the length of the string. Subsequent command words contain information in multiples of four bits of data. The following are the types of GPS and their parameters:

comment **cw** [**string**]

cw is the control word. The first four bits identify the **comment** primitive and have the value 0xF. The following bits give the command word count for the primitive.

[**string**] is a string of characters terminated by a null character. If the string does not end on a command word boundary, another null character is added to align the string with the command word boundary.

lines **cw points sw**

cw is the control word. The first four bits identify the **lines** primitive and have the value 0x0. The remaining bits give the command word count for the primitive.

points is one or more pairs of integer coordinates having values

within a Cartesian plane or universe of 65,536 points on each axis (-32,768 to +32,767).

sw is the style command word. The first eight bits hold an integer value for **color** information. The next four bits contain an integer value for **weight** to indicate line thickness:

0 Narrow
1 Bold
2 Medium.

The last four bits of **sw** specify an integer value giving line **style** information:

0 Solid
1 Dotted
2 Dot-dashed
3 Dashed
4 Long dashed.

arc **cw points sw**

cw is the control word. The first four bits identify the **arc** primitive and have the value 0x3. The next twelve bits contain the command word count for the primitive.

points is one or more pairs of integer coordinates having values within a Cartesian plane or universe of 65,536 points on each axis (-32,768 to +32,767).

sw is the style command word. The first eight bits are an integer value for **color**. The next four bits contain an integer value for **weight** to indicate line thickness:

0 Narrow
1 Bold
2 Medium.

The last four bits contain an integer value setting line style:

0 Solid
1 Dotted
2 Dot-dashed
3 Dashed
4 Long dashed.

text **cw point fw so** [**string**]

cw is the control word. The first four bits identify the **text** primitive and have the value 0x2. The remaining twelve bits contain the command word count for the primitive.

point is a pair of integer coordinates that are a value within a Cartesian plane or universe of 65,536 points per axis (-32,768 to +32,767).

fw is a font command word. The first eight bits contain an integer value for **color** information. The next eight bits contain an integer value for **font** information, with four bits giving the **weight** (density) value for the font, and four bits

giving the **style** (typeface) value for the font.

so is a size/orientation command word. Eight bits specify **textsize** as an integer value to indicate the size of characters drawn. **textsize** represents character height in absolute **universe units**. The actual character height is five times the **textsize** value. The next eight bits are a signed integer value for **textangle**, and express the angle and direction of rotation of the character string around the beginning **point**. **textangle** is expressed in degrees from the positive **x** axis. The **textangle** value is 256/360 of its absolute value.

hardware **cw point** [**string**]

cw is the control word. The first four bits identify the **hardware** primitive and have the value 0x4. The next twelve bits indicate the command word count for the primitive.

point is a pair of integer coordinates that are values within a Cartesian plane or universe of 65,536 points on each axis (-32,768 to +32,767). This **point** is the starting point for the **string**, which is a string of hardware characters or control commands to a hardware device.

Related Information

In this book: "stat.h" in topic 2.4.22.

2.3.26 group

Purpose
Identifies a group.

Description
Users can be assigned to one or more groups, each of which share certain protection privileges. The person who sets up the system may want to place users in the same group because they need access to a common set of files. Similarly, a certain group of users can have access restricted to certain files.

When users log in, they are assigned to the group specified in the **/etc/password** file. In addition, they are assigned as a member of all groups specified in this file. Users are allowed to access any files that the group to which they are assigned has access. However, any files created by the user can be accessed only by the members of the primary group of which that user is a member. A user is allowed to change his primary group for the duration of the terminal session using the **newgrp** command.

The **/etc/group** file defines to which groups a user has membership. Each line in this file defines a group and consists of four fields separated by colons. It contains the following information for each group:

Note: In the multibyte environment, the **group** file may contain only ASCII characters.

Group	Description
group name	A character string of up to eight characters that references the group.
password	This field is optional. If specified, anyone attempting to enter the group must correctly supply the password to the system.
group ID	A number assigned to the group and used in access decisions.
user group list	A list that specifies the login names of all users allowed in the group. User IDs in the list are separated by commas. The user group list may contain up to 500 eight-character login names.

In newly distributed systems, there are typically only two groups: the staff group and the system group. New users can be added to groups and new groups can be added as necessary.

If several users wish to share the same privileges, including the ability to terminate each other's processes as well as to access the files of others, the same numerical user ID can be assigned to each. This mechanism is sometimes used to give the same person several accounts on the system, each with potentially different login directories and other characteristics, such as electronic mailboxes or login programs. For example, the operator has the same user ID, and therefore superuser authority. However, this operator typically uses a restricted version of the shell that does not give access to commands that allow reading the files of others.

Example

The following is an example of the **/etc/group** file. This is an ASCII file. Each group is separated from the next by a new-line character. The fields are separated by colons. Because the password is encrypted, it can be used to map numerical group IDs to names without concern of compromise to user security.

```
system::0:su,bill,jack,gary
staff::1:
bin::2:su,bin
sys::3:su,bin.sys
adm::4:su,bin,adm
mail::6:su
usr::100:guest
```

File

/etc/group

Related Information

In this book: "passwd" in topic 2.3.44.

The **newgrp, passwd,** and **users** commands in *AIX Operating System Commands Reference*.

2.3.27 history

Purpose
Contains the history of an installed licensed program product.

Description
Each licensed program or component of a licensed program that is shipped by IBM contains a history file. The purpose of a history file is to identify the installed release and version of a licensed program or component and to provide a record of any updates (level changes). A history file is replaced when a component is reinstalled. History files for programs installed on the operating system are named **/usr/lpp/pgm-name/lpp.hist**, where **pgm-name** is the name of the licensed program or component.

The history file consists of a series of 80-character records. The first two records contain the install data and all subsequent records contain update data. There are three different formats of 80-character records:

Record	Description
Information	Identified by an **a**, **c**, **r**, or **v** character in position 1. The **install** and **update** procedures use information records to identify the licensed program or component name; the current version, release, and level; the date the record was added; and the user who initiated the install or update. Figure 3-3 shows the format of the fields in the information records.
Title	Identified by a **t** in character position 1. Contains the descriptive title (up to 30 characters) for the licensed program or component, starting in character position 3. The title record must always be the second record in the history file.
Comment	Identified by an ***** (asterisk) in character position 1. Allows descriptive comments to be entered into the history file. An ***** is usually placed in character position 79 to ensure a full 80-column record.

The last character of each record (character position 80) must be a new-line character. Unused character positions must be blank-filled. Tab characters are not permitted.

The first record in a history file must be an information record with a **c** in character position 1. The second record must be the title record. These two records contain data about the installation of the program. The remaining records in the file may be any combination of information and comment records, and they identify updates to the program.

Figure 3-3 shows the format of an information record in the history file. The definitions for each of the fields other than character position 1 are explained following the figure.

Figure 3-3. Information Record Format

Location	Field	Description
1	.S	The type of information record:

		a Indicates that the update has been applied. **c** Indicates that the update or install has been committed (accepted). **r** Indicates that the update has been rejected.
3-10	pgm-name	The name assigned to the program (lowercase characters only). If the name contains less than 8 characters, it must be padded with blanks.
11-17	reserved	Reserved
18-20	VV.	A 2-digit numeric field followed by a period indicating the version number of the program. The version number indicates the level of the hardware and operating system with which the program works.
21-23	RR.	A 2-digit numeric field followed by a period indicating the release number of the program. The release number tracks changes to external programming interfaces since the last version change. This number is generally incremented each time the external interface to the program changes.
24-28	LLLL.	A 4-digit numeric field indicating the update level of the program. This field is incremented when the changes to the program do not affect external programs that may use the documented external interface for the program. The level, together with the **S** field, ensures that all changes up to and including the current change are installed on the system. The fourth (or units) digit of the level is normally 0. IBM reserves this digit for future use.
29-32	FFFF	A 4-digit numeric field indicating the fix number applied to the program. A fix number of 0 denotes an upgrade; all other values denote fixes.
33	blank	Blank
34-39	DDMMYY	These three numeric fields indicate the date the program changed: *DD* Day of the month (1 to 31). *MM* Month of the year (1 to 12). *YY* Year (00 to 99).
40	blank	Blank
41-48	username	An alphanumeric field that contains the user name of the person who installed the program.

		If the user name is shorter than eight characters, it must be padded with blanks.
49	blank	Blank
50-79	comment field	A 30-character field for comments. An ***** (asterisk) is usually placed in character position 79 to ensure a full 80-column record.
80	**\n**	A required new-line character, indicating the end of the record.

Each cluster site also has a local history file that is used to log relevant information on changes made to the local file system on that site. This file is called **/local/lpp.hist**. Local history files are of interest to the system administrator as a log of each site's past. They are not used for determining the local changes pending for a site or as a means of controlling programs; these functions are performed by the queue mechanism.

The format for the local history file is identical to the format described above.

File

/usr/lpp/pgm-name/lpp.hist

Related Information
The **installp** and **updatep** commands in *AIX Operating System Commands Reference*.

2.3.28 inittab

Purpose
Describes the information used by the **init** process.

Description
The **inittab** file supplies the script to **init**'s role as a general process dispatcher. The process that constitutes the majority of **init**'s process dispatching activities is the line process **/etc/getty** that initiates individual terminal lines. Other processes typically dispatched by **init** are daemons and the shell.

Note: If the Transparent Computing Facility is installed, each cluster site has its own version of **inittab** (**/etc/inittab** is a symbolic link into the local file system).

Subtopics
2.3.28.1 File Format
2.3.28.2 inittab Parameters

2.3.28.1 File Format

The **inittab** file consists of one or more named stanzas separated by blank lines. Each stanza begins with its name followed by a colon, and contains assignments of values to keyword attributes. The values, in turn, may be alphanumeric strings or arbitrary character strings enclosed in double quotes.

Note: In the multibyte environment, the **inittab** file stanzas may contain only ASCII characters.

2.3.28.2 inittab Parameters

The **inittab** keywords and their meanings are as follows:

id This is one to four characters used to uniquely identify an entry.

level This defines the run-level in which this stanza is to be processed. Run-levels effectively correspond to a configuration of processes in the system. That is, each process spawned by **init** is assigned to a run-level or run-levels in which it is allowed to exist. The run-levels are represented by a number ranging from 0 through 6. As an example, if the system is in run-level 1, only those stanzas having a 1 in the **level** keyword are processed. When **init** is requested to change run-levels, all processes which do not have an entry in the **level** keyword for the target run-level will be sent the warning signal (**SIGTERM**) and allowed a 20-second grace period before being forcibly terminated by a kill signal (**SIGKILL**). The **level** keyword can define multiple run-levels for a process by selecting more than one run-level in any combination from 0 through 6. If no run-level is specified, then **action** will be taken on this **process** for all run-levels 0-6. There are three other values, a, b, and c, which can appear in the **level** keyword, even though they are not true run-levels. Stanzas which have these characters in the **level** keyword are processed only when the telinit (see the **init** command in *AIX Operating System Commands Reference*) process requests them to be run (regardless of the current run-level of the system). They differ from run-levels in that the system is only in these states for as long as it takes to execute all the stanzas associated with the states. A process started by an a, b or c command is not killed when **init** changes levels. They are only killed if their stanza in **/etc/inittab** is marked off in the **action** keyword, their stanza is deleted entirely from **/etc/inittab**, or **init** goes into the SINGLE USER state.

action Values for this keyword tell **init** how to treat the process specified in the **command** keyword. The actions recognized by **init** are as follows:

respawn If the process does not exist then start the process, do not wait for its termination (continue scanning the **inittab** file), and, when it dies, restart the process. If the process currently exists, then do nothing and continue scanning the **inittab** file.

wait Upon **init**'s entering the run-level that matches the stanza's **level**, start the process and wait for its termination. All subsequent reads of the **inittab** file while **init** is in the same run-level will cause **init** to ignore this stanza.

once Upon **init**'s entering a run-level that matches the stanza's **level**, start the process, do not wait for its termination and when it dies, do not restart the process. If upon entering a new run-level, where the process is still running from a previous run-level change, the program is not restarted.

boot The stanza is to be processed only at **init**'s boot-time read of the **inittab** file. **init** is to start the process, not wait for its termination, and when it dies, not restart the process. In order for this instruction to be meaningful, the **level** should be the default or it must match **init**'s run-level at boot time. This action is useful for an initialization function following a hardware reboot of the system.

bootwait The stanza is to be processed only at **init**'s boot-time read of the **inittab** file. **init** is to start the process, wait for its termination and, when it dies, not restart the process.

powerfail Execute the process associated with this stanza only when **init** receives a power fail signal (**SIGPWR**); see "sigaction, sigvec, signal" in topic 1.2.263.

powerwait Execute the process associated with this stanza only when **init** receives a power fail signal (**SIGPWR**) and wait until it terminates before continuing any processing of **inittab**.

off If the process associated with this stanza is currently running, send the warning signal (**SIGTERM**) and wait 20 seconds before forcibly terminating the process via the kill signal (**SIGKILL**). If the process is nonexistent, ignore the stanza.

ondemand This instruction is really a synonym of the respawn action. It is functionally identical to respawn but is given a different name in order to divorce its association with run-levels. This is used only with the a, b or c values described in the **level** keyword.

initdefault A stanza with this **action** is only scanned when **init** is initially invoked. **init** uses this entry, if it exists, to determine which run-level to enter initially. It does this by taking the highest run-level specified in the **level** keyword and using that as its initial state. If the **level** keyword is empty, this is interpreted as 0123456 and so **init** will enter a run-level 6. Also, the stanza with the **initdefault** action can specify s or S in the **level** keyword to indicate that **init** is to start in the SINGLE USER state. Additionally, if **init** doesn't find an **initdefault** stanza in **/etc/inittab**, then it will request an initial run-level from the user at reboot time.

sysinit Stanzas with this action are executed before **init** tries to access the console. It is expected that this stanza will be used to initialize devices on which **init** might try to ask the run-level question. These stanzas are executed and waited for before continuing.

command This is an **sh** command to be executed.

comment The text that follows is a comment. If the text includes a space (you have two or more words), then the test must be quoted. The commands **WHO** *-l* and **WHO** *-u* print the comment.

Example
The following example of an **inittab** file illustrates some of its features:

```
*
idef:
     id = idef
     level = 2
     action = initdefault
     command = "# default state: multi user"
*
sys1:
     id = sys1
     level = NULL
     action = sysinit
     command = "/etc/init.dir/shx Boot2singl"
*
mul1:
     id = mul1
     level = 2
     action = wait
     command = "/bin/kill -9 -1"
*
mul2:
     id = mul2
     level = 2
     action = wait
     command = "/etc/init.dir/shx Singl2multi"
*
sin1:
     id = sin1
     level = 03
     action = wait
     command = "/etc/init.dir/shx Multi2singl"
*
sin2:
     id = sin2
     level = 03
     action = wait
     command = "/bin/kill -15 -1"


*
sin3:
     id = sin3
     level = 03
     action = wait
     command = "sleep 10"
*
*
sin4:
     id = sin4
     level = 03
     action = wait
```

```
     command = "/bin/kill -9 -1"
*
sin5:
     id = sin5
     level = 03
     action = wait
     command = "/etc/init.dir/shx Filesystems down"
*
sin6:
     id = sin6
     level = 3
     action = wait
     command = "/etc/init s </dev/console >/dev/console 2>&1"
*
hlt0:
     id = hlt0
     level = 0
     action = wait
     command = "/etc/haltsys"
*
lds:
     id = lds
     level = 4
     action = respawn
     command = "/etc/loadserver"
*
console:
     id = con
     action = respawn
     level = 14
     command = "/etc/getty console"
*
tty00:
     id = tt0
     action = respawn
     level = 14
     command = "/etc/getty tty00"
*
*
tty01:
     id = tt1
     action = off
     level = 14
     command = "/etc/getty tty01"
     comment = "modem line"
```

Files

/etc/inittab
/etc/locks

Related Information

In this book: "attributes" in topic 2.3.5, "connect.con" in topic 2.3.9, "environment" in topic 2.4.6, and "termio" in topic 2.5.28.

The **su, pstart, pdisable, getty, login, init**, and **stty** commands in *AIX Operating System Commands Reference*.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

2.3.29 inode

Purpose
Describes a file system file or directory entry as it appears on a disk.

Synopsis

```
#include <sys/types.h>
#include <sys/ino.h>
```

Description
An **inode** for an ordinary file or directory in a file system has the following structure defined by **sys/ino.h**:

```
      /* Inode structure as it appears on a disk block. */

  #define NADDR   13
  #define SMBLKSZ 384      /* bytes available in the inode for data */

  struct  dinode
  {
   unsigned short  di_mode;           /* mode and type of file */
   short           di_nlink;          /* number of links to file */
   short           di_uid;            /* owner's user id */
   short           di_gid;            /* owner's group id */
   ino_uniqid_t    di_size;           /* inode reuse count */
   short           di_filler;         /* filler */
   dflag_t         di_dflag;          /* disk flags */
   off_t           di_uniqid;         /* unique identifier */
   time_t          di_mtime;          /* time last modified */
   time_t          di_atime;          /* time last accessed */
   time_t          di_ctime;          /* time changed */
   commitcnt_t     di_cmtcnt;         /* gfs commit seq number */
   fstore_t        di_fstore;         /* file propagation attributes */
   long            di_version;        /* version number this copy of data */
   daddr_t         di_blocks;         /* actual number of blocks used */
   char            di_pad[27];        /*reserved for future use */
   char            di_sbflag;         /* flags for small blocks */
   daddr_t         di_addr[NADDR];    /* disk block addresses */
   char            di_sbbuf[SMBLKSZ]; /* small block buffer */
  };
```

The fields in the structure are as follows:

di_addr Array of thirteen 4-byte block numbers assigned to this file. The first ten block numbers are direct addresses while the last three are indirect addresses.

di_atime Time this file was last accessed.

di_blocks The number of file system blocks used by this file.

di_cmtcnt Commit sequence number; assigned when the file is committed. Replicated files are propagated to other sites in the order in which they were committed.

di_ctime Time this file or inode was last changed.

di_dflag Disk flags. DISOCKET, DIHIDDEN, and DILINK are used with

flags in **di_mode** to define additional file types. Other flags are:

DIDEL File has been deleted (the inode is unallocated).

DIALLOC The file is currently allocated.

DILONGDIR File is a directory with variable-sized entries. This flag is set for all directories.

DISTORE The file is stored. In a non-replicated file system, this flag is set in the same instances as DIALLOC. In a replicated file system, this flag indicates that the file is stored by this copy of the file system.

di_fstore File storage attribute. Used to determine on which sites a file in a replicated file system is stored. This value is not used in a non-replicated file system.

di_gid Group ID.

di_mode Type and access permissions of file. This field is encoded identically to the **st_mode** field returned by the **statx** system call, with the following exceptions:

Sockets are encoded with IFIFO in **di_mode** and DISOCKET in **di_dflag**, rather than S_IFSOCK.

Hidden directories are encoded with IFDIR in **di_mode** and DIHIDDEN in **di_dflag**, rather than S_IHIDDEN.

Symbolic links are encoded with IFREG in **di_mode** and DILINK in **di_dflag**, rather than S_IFLNK.

di_mtime Time this file was last modified.

di_nlink Number of directory entries that name this file.

di_sbflag Has a nonzero value when the file is no more than SMBLKSZ bytes in size. It indicates that the entire file content can be found in **di_sbbuf**. All entries in **di_addr** are 0 when **di_sbbuf** is in use.

di_sbbuf The buffer to store the file content when **di_sbflag** is nonzero.

di_size Number of bytes in the file.

di_uid Owner ID.

di_uniqid Inode reuse count. This is incremented on file creation. Used to determine if a file has been deleted and then had its inode reused. This field corresponds to the **st_gen** field returned by **statx**.

di_version A count of the number of times a file has been updated.

See the **types.h** file for related information concerning the types used in **struct dinode**.

Related Information
In this book: "fs" in topic 2.3.20, "stat.h" in topic 2.4.22, and "types.h" in topic 2.4.27.

2.3.30 kaf

Purpose
Specifies how to process **ddi** keywords and their parameters.

Description
Keyword Attribute Files, also called **kaf** files, define how the **devices** command and customize helpers are to process keywords used in **ddi** files. The **kaf** files:

Contain instructions for processing device informatio
Control whether the **devices** command displays the associated information
Control whether a user can change the information using the **devices** command
Specify the input validation that the **devices** command performs
Determine the action that the customize helper takes

The **kaf** information can be included in the **ddi** file for the device, or it can appear in a separate file. If it is contained in a separate file, then the stanza for the device in the **system** file must name the **kaf** file as the value of the **kaf_file** keyword. The **kaf_use** keyword (also in the **system** file) specifies the stanza of the **kaf** file to use.

The name of each stanza in a **kaf** file is the name of a keyword that is used in **ddi** files. The stanza defines how the **devices** command and customize helper programs process that **ddi** keyword. The following section defines the keywords that can appear in the stanzas of **kaf** files.

The use of extended characters in **kaf** files is not supported.

Devices that can be added, deleted, or changed by **devices** must have a **kaf** file.

Subtopics

2.3.30.1 Control over Display and Modification of the Keyword

display If set to TRUE, then the **devices** command displays the device characteristic keyword and allows the user to change its value.

2.3.30.2 User Input Validation

vtype Specifies the type of checking that the **devices** command performs on values entered by the user. **vtype** can be set to one of the following values:

- **0** No validation.
- **1** Mapping validation: The input value must be one of the keywords found in the stanza named by the **map** keyword.
- **3** Range validation: The input value must have the data type specified by the **type** keyword and must fall in the range specified by the **range** keyword.

map Names a stanza in the **kaf** file that contains a list of **keyword=value** pairs against which the input value is to be matched. If the input matches a given **keyword**, then the corresponding **value** is substituted in its place.

opts Specifies the **option**s to search for in the **/etc/ddi/options** file. **opts** is one of the following:

- **k** Keyword only
- **c** Keyword followed by device class
- **t** Keyword followed by device type
- **s** Keyword followed by device stanza name.

If the **opts** keyword is not specified, its value defaults to **k**. See "options" in topic 2.3.43 for details about the **/etc/ddi/options** file.

range Defines the valid range of values for a keyword so that the **devices** command can verify values entered by the user. The value of the **range** keyword has the format **first,last,incr**, where **first** is the first number in the range, **last** is the last number, and **incr** is the increment between values in the range. For example, **range=2,10,2** specifies the values 2, 4, 6, 8, and 10.

type Defines the data type for the value of a keyword. The **devices** command ensures that the values entered are the correct data type, specified by one of the following:

- **F** Floating-point (**float**)
- **H** Hexadecimal (**int**)
- **I** Integer (**int**)
- **L** Long integer (**long int**)
- **S** Short integer (**short int**)
- **U** Unsigned integer (**unsigned int**).

Files

/etc/ddi/opprinter.kf
/etc/ddi/osprinter.kf
/etc/ddi/plotter.kaf
/etc/ddi/pprinter.kaf
/etc/ddi/sprinter.kaf
/etc/ddi/tty.kaf
/etc/mdkaf
/etc/ddi/c327
/etc/ddi/nty.kaf
/etc/ddi/pty

Related Information
In this book: "attributes" in topic 2.3.5, "ddi" in topic 2.3.13, "descriptions" in topic 2.3.14, and "system" in topic 2.3.56.

2.3.31 loads

Purpose

Contains loads information on all sites. **/etc/loads** is read by the **fast**, **fastsite**, and **loads** commands.

Description

The **/etc/loads** file contains loads and user information for all sites. It includes, for each site, the site number, the site name, the number of users, the length of time in seconds the site has been up, and 1 minute, 5 minute, and 15 minute load averages.

The **loadserver** on each site updates **/etc/local_loads**, and in each cluster, one **loadserver** is responsible for extracting local load and user information and updating **/etc/loads**.

Files

/etc/loads
/etc/local_loads

Related Information

The **loads** and **loadserver** commands in *AIX Operating System Commands Reference*.

2.3.32 master

Purpose
Contains master configuration information.

Description
The **master** file is an attribute file that contains stanzas that describe all device drivers and the system parameters that could be configured into the system.

There are two general kinds of stanzas in the master file. They are:

AIX device driver stanzas describing attributes relevant to th configuration of the drivers.

Stanzas describing system parameters that are relevant to the syste configuration.

The **type** keyword in each stanza identifies the kind of stanza it is. AIX driver stanzas specify drivers to link into the kernel.

The use of extended characters in the **master** file is not supported.

Subtopics

2.3.32.1 AIX Driver Stanzas

A unique set of keywords is associated with each type of stanza. It is not necessary, however, for a stanza to contain all the keywords associated with that type. If a keyword is omitted from the stanza, the default is used. Mandatory keywords must be supplied and are not defaulted. The name of each stanza is a logical AIX driver name referenced in other stanzas.

The lines interpreted by the **config** and **osconfig** commands are:

block Indicates if the device driver supports block interface (TRUE or FALSE).

char Interpreted by the **config** program. **char** has the same interpretation as **character**; that is, the device is a CHARACTER device as opposed to a BLOCK device.

config Indicates that this device has a customization helper program, which provides assistance in decoding other options. This value is the name of the helper program in the **/etc** directory.

devhdrreqd
Used by the **config** program. If TRUE, **config** adds the information listed below to the **conf.c** file it builds. Note that this keyword should not be used in PS/2 systems unless the driver is included in the base system provided, since the **conf.c** file is not compiled.

If the device is a BLOCK device, a **#define B_hndlr devmajor** where **hndlr** is replaced with the value from the **prefix** keyword and **devmajor** is replaced with the value from the **major** keyword.

If the device is a CHARACTER device, a **#define C_hndlr devmajor** where **hndlr** is replaced with the value from the **prefix** keyword and **devmajor** is replaced with the value from the **major** keyword.

If the device is neither a BLOCK nor a CHARACTER device, a **#define M_hndlr devmajor** where **hndlr** is replaced with the value from the **prefix** keyword and **devmajor** is replaced with the value from the **major** keyword.

A **#define U_hndlr naddrs** where **hndlr** is replaced with the value from the **prefix** keyword and **subunits** is replaced with the count of the configured subunits (not the keyword) for the given device.

A **#define N_hndlr naddrs** where **hndlr** is replaced with the value from the **prefix** keyword and **naddrs** is replaced with the number of configured addresses for the device.

A **#include hndlconf.h** where **hndlr** is replaced with the value from the **prefix** keyword.

devtable Used by the **config** program. If TRUE, **config** adds the information listed below to the **conf.c** file it builds. Note that this keyword should not be used in PS/2 systems unless the PS/2 driver is included in the base system provided, since the **conf.c** file is not compiled.

An initialized declaration for **struct dev_unit hndlrdevs[]**, where **hndlr** is replaced with the value from the **prefix** keyword. This structure is initialized with values taken from the device, including **stanzaname**, and the **driver**, **units**, and **address** keywords.

If **devtable** is TRUE, then the name of the handler table is output into the appropriate slot in the **gensw** table for the device. If **devtable** is false, this slot initializes with a zero.

major Identifies the major device number for this driver. This is mandatory.

mandatory Identifies this driver to be included into the AIX system kernel whether or not the **system** file asks for it. If this value is TRUE, include this driver.

maxminor States the maximum number of minor devices this driver supports. This number should agree with the driver code.

mpx Identifies a multiplexed special file when this value is TRUE.

prefix Provides a prefix for the driver routines. For example, if this value is **abc**, then the **open** routine in the driver is **abcopen**. This keyword is mandatory. Note that all drivers are assumed to be archived into the system object libraries.

routines Identifies the routines actually defined for this driver. The possible routines are **open**, **close**, **read**, **write**, **strategy**, **ioctl**, **init**, and **select**.

struct Indicates the name of the **iobuf** structure associated with a block-type device.

subunits This keyword is not used.

type Specifies the type of stanza. The possible values for device drivers are:

alias Allows two names for the same device driver.

dev Indicates that this device is associated with an adapter.

sw Indicates that this device is not associated with hardware, for example, the pseudo-device driver.

tty Specifies that this stanza is a line discipline device driver.

type=linedisc
Provides the same attributes as **type=tty;**.

block=TRUE:
Interpreted by the **config** program. This causes **struct iobuf hndlrtab** entries to be generated in the **conf.c** file, with their addresses initialized in **devsw**. Indirectly causes the **hndlrstragety** entry points to be generated in the **conf.c** file, with their addresses initialized in **devsw**. Causes the **b** entries

in the device types summary section of the configuration summary file. Causes **block** or **bl/ch** entries in the block and character device tables section of the configuration summary file.

nocount=TRUE:
Interpreted by the **config** program. Suppresses the declaration of **int hndlr_cnt = numunits**. Normally output to the **conf.c** file, where **hndlr** is replaced with the value from the **prefix** keyword and **numunits** represents the number of configured devices of that type.

oneonly=TRUE:
Interpreted by the **config** program. Causes an error to be generated if more than one stanza for this device is encountered.

line=n Interpreted by the **config.** program. In stanzas with **type=tty** or **type=linedisc**, this is used to indicate the position within **linesw**, which this line discipline is to occupy.

The line interpreted by **crash** and **pstat** is:

ttyflg=alloc,ttysum,cntsym:
If a device supports **tty** output and thus maintains a **tty** structure, the programs **crash** and **pstat** need to be able to read these structures. They read the **/etc/master** file for information on how these structures are allocated. The value of **alloc** describes the method the driver uses to allocate the **struct tty**:

```
"static"  -> struct tty xx_tty[COUNT];
"static*" -> struct tty *xx_tty[COUNT];
"auto"    -> struct tty *xx_tty;
"auto*"   -> struct tty *xx_tty;
```

The **ttysym** is the symbol name of the **tty** structure as declared in the device driver.

The **cntsym** is the symbol name of the number of tty structures present, as declared in the device driver.

2.3.32.2 System Parameter Stanzas

System Parameter Keywords:

The following is a list of keywords used to define system parameters in the **master** file:

default The default value of the system parameter.

patchaddr The address in the kernel which stores the value of the parameter.

patchlen The parameter's length in bytes in the kernel.

symbol The name of the system parameter as it appears temporarily in the defines of **/tmp/sysgen.<machine-id>/conf.c**; this file is removed during system build.

type The type of the stanza. The following are the valid values of **type**:

- **parm** System parameters.
- **special** Special system parameters used in system configuration.
- **text** The file specified by **default** is included in **conf.c** when **/etc/config** is run.
- **udev** Special files like **rootdev** are for the system device.

System Parameters:

The following are object-code options for the PS/2 only:

ipc43 Includes TCP/IP code in the kernel.

kerndbg Includes kernel debugger code in the kernel.

merge Includes the DOS Merge code in the kernel.

nfserver Includes the NFS code in the kernel.

Note: All system parameters that are listed in **/etc/master/** may have their default values overriden by the values in **/etc/system**.

2.3.32.3 Site-Specific Parameters

Note: Stanzas which do not have the **patchaddr** attribute cannot be patched. The parameters **dump**, **dumplow**, **nswap**, **pipe**, **root**, **swap**, **swaplow**, **swapmap**, and **swbufs** are automatically configured by the kernel. However, they can be overridden in **/etc/system**.

bhash Buffer hash group size.

buffers Buffers are used to provide the means to regroup I/O requests to a block device. Often user and file system block sizes differ, and it is more efficient to perform the I/O in an increment that optimizes slow device access. So when writing to a block device, for example, the kernel buffers the data until a convenient amount has been stored for writing all at once to the device. Also, buffers allow data to be cached so that disk I/O can be reduced. Normally this parameter is set to 0 and at system run-time 20% of free memory is allocated to buffers. This is bounded by the **minbufs** and **maxbufs** parameters. However, by changing this value, these calculations are not performed, and the user supplied value is used instead.

callout Callouts are used by the kernel as a means of performing some task after a certain amount of time. It is possible that a user, upon purchasing a new device driver, could cause the system to reach its upper limit on the number of callouts allowed. Should this happen, the system would panic with **Timeout table full**. If the new functionality added to the system is error-free, increasing the number of callouts (which cost 24 bytes each) could solve the problem.

clist **Clist**s (character lists) transfer data between user input and the kernel. If there aren't enough **clist**s, users notice long delays in even simple character echos. If there are a lot of users on the system, try increasing the number of **clist**s. It costs 74 bytes per extra member.

cluster_id
Specifies the cluster ID in the Internet Protocol (IP) format (a.b.c.d.).

daylight Nonzero if daylight savings time in effect.

dcbuffs, dchash
Directory cache buffers are an optimization for performing directory searches. For example, in the command **cd /foo/bar**, the following happens: in the directory /, the entry **foo** is searched in a linear fashion. Once the entry is found, its information is stored in a directory cache buffer on the theory that if a user needed that information once, he may need it again soon. The cache eliminates the need to linearly search the directory for the entry **foo** if it is referenced in the near future. To access the directory cache buffer efficiently, a directory hash table is used. The **dchash** parameter indicates the number of directory hash groups. The smaller the number, the larger the size of the hash table. Increasing the number of buffers could cause a performance increase in directory access. Each buffer costs 52 bytes plus any increase in the size of the hash table.

depsite Indicates that the kernel is to run on a site that does not store a copy of the **root** file system.

dump Kernel core images (called a dump) are made when the system stops working. On systems with a dump minidisk located on the hard disk, the kernel auto-configures the dump properly. The user can change the dump device, by placing an entry in the system file and rebuilding the kernel using the **newkernel** command. The **dump** parameter should reference any unused portion of a minidisk.

dumplow Offset in the dump minidisk to begin dumping a kernel image. This offset is in 4K blocks. Normally the value of the **dumplow** parameter is 0, but it can be changed to allow dumping to an arbitrary point on a minidisk. This could allow for the dump minidisk to overlap with another minidisk. The **dumplow** parameter is then set to last used block of the minidisk.

errsize The size (in bytes) of the error buffer used for error logging. If error logs are wrapping too much, increasing this number helps.

files This is the number of entries in the file table. The file table contains 1 entry per open file access in the system. If users find that programs are failing with **ENFILE**, the system administrator should check **/dev/osm** to discover which table overflowed. If it contains **File table full at 'n' entries**, try increasing the value of the **files** parameter. The cost is 20 bytes per entry.

generic Indicates an installation-mode kernel.

ghash Specifies the optimum length of the hash chain for the mount table. Typically, the smaller the **ghash** parameter, the better the performamce.

gmounts Maximum number of mounted filesystems in cluster.

gpgscln Get pages clean. When the number of free pages on the swap device plus the number of free memory pages divided by 2 is less than this limit, the kernel removes unreferenced pages from working sets of the process.

gpgslo, gpgshi
Get pages low/high. These are low and high water marks indicating when to begin stealing memory from processes. If there is a lot of free memory available, AIX waits until **gpgslo** free pages are remaining before beginning to page out. Page out continues until **gpgshi** is reached.

hz Frequency of hardware clock interrupts.

ihash Inode (incore inodes) hash group size.

killem When the amount of free swap space plus free memory space divided by 2 reaches this threshold, the kernel begins killing processes to free more swap space and memory.

The **gpgscln** and **killem** parameters are used to calculate three additional parameters (internal to the kernel):

danger = **gpgscln** - ((**gpgscln** - **killem**) / 3)
nofork = **danger** - ((**gpgscln** - **killem**) / 3)
killcur = **killem** / 2

Altogether, these five parameters indicate to the kernel what actions to take when the number of free pages falls below the specified value:

gpgscln: Get pages clean.

danger: Send **SIGDANGER** to all processes.

nofork: Disallow nonsuperuser forks.

killem: Kill processes with a large number of pages.

killcur: Kill current process.

locks These are responsible for locking files. Should the number of locks be too small, applications such as **passwd** will fail because the **lockf** system calls will fail. Increasing the number of locks costs 32 bytes each.

locsite Uniquely identifies a TCF cluster site.

maxbufs The maximum number of buffers that the system allocates.

maxinod,mininod
The inode table (described by its minimum and maximum values) contains **mininod**- one entry per open file in the system. This is different from the **files** parameter in that, while a file may have multiple entries in the file table, there is only one entry in the inode table. If the system administrator checks **/dev/osm** and notices **Inode table full at 'n' entries**, then he should try increasing the maximum number of inodes. The kernel begins allocating **mininod**s and progressively more (as needed) until **maxinod** is reached.

maxproc Maximum number of processes per user. If a user exceeds this limit, he won't be able to run any more programs. It keeps one user from tying up an entire system's resources.

minbufs The minimum number of buffers that the system allocates.

mounts Not used.

msfiles Maximum swap minidisks allowed.

msgmax The maximum message size that can be transmitted.

msgmnb The maximum bytes allowed on the message queue.

msgmni The number of message queue identifiers.

msgssz Message segment size.

msgtql Number of system message headers.

name Each machine has a name associated with it.

net Contains the major number of the TCF Network File System.

nfs Contains the number of the NFS logical device.

nmasz Target number of **netmsgs** per allocation.

nnetmsg Maximum number of **netmsgs** in system.

npacks Number of **packlist** structures.

npbuf Number of **physio** buffers. These buffers are used when swapping out processes and copying them between memory and the swap device.

nswap Size (in 4K blocks) of the swap minidisk. Together with the **swaplow** parameter, these can describe any arbitrary region of the swap device.

ntyunits The maximum number of AIX Access for DOS sessions that the host machine can handle at once. The default value is 8. Allowing one additional session costs 64 bytes plus 444 bytes once it's used.

osmsize Size of the **osm** buffer (in bytes).

phash Process hash group size.

pipe The bootable AIX minidisk containing the kernel.

procs Each process on the system has an entry in the process table. If there are many simultaneous tasks on the system, the process table could overflow. This creates a situation where users would often find that any command entered does not execute. Each member of the process table costs 224 bytes. However, there are many other parameters which are allocated based on the size of the process table, and so of course they will grow too.

props Size of propagation table.

ptrace Number of available local **ipc** structures.

pvsegs The number of process **vsegs** is based on this value. The formula used is (number of processes * the **pvsegs** parameter). AIX assumes an average of 6 **pvsegs** per process (**pvsegs** has a 32-byte overhead).

ptyunits The maximum number of pseudo-terminal login sessions that the host machine can handle at once. The default value is 8. Allowing one additional session costs 68 bytes plus 380 bytes once it is actually used.

root The minidisk where the **root** file system is located.

rprocs Relevant to number of remote processes this site originates.

rsleep Remote sleep records.

rwake Remote wakeup records.

semaem Maximum value of the adjust on exit status.

semopm Maximum number of operations per **semop** system call.

semume Maximum number of undo entries per process.

semvmx Maximum value of a semaphore.

semmni Number of semaphore identifiers.

semmns Upper limit on number of semaphores in the system.

semmsl Maximum number of semaphores per semaphore ID.

shash Sleep hash group size.

These parameters control the size of their respective hash tables. If the hash tables are often empty, then the hash group sizes could be increased, thereby freeing more memory. However, if the tables are continually full, decreasing the hash group sizes increases the size of the hash tables.

shmbrk The gap (in clicks) between data and shared memory.

shmseg Maximum number of shared memory segments per process.

shmmax Maximum shared memory segment size.

shmmin Minimum shared memory segment size.

shmmni Number of shared memory identifiers. The upper limit of the **shmmni** parameter is dependent upon the **procs** parameter as defined in the following formula:

1 < = **shmmni** < = 4 * **procs** - 50

By default, the **procs** parameter is set to 74. This can be verified in the **procs** stanza in the **/etc/master** file and the definition of users can be noted in the **sysparms** stanza of the **/etc/system** file. The **shmmni** parameter has the following range:

1 < **shmmni** < 250

To use more than 250 shared memory segments, increase the value of the **procs** parameter.

sites Maximum number of possible sites in network.

sphiwat Idle count that causes sever processes to shut down.

splowat Idle count that causes more sever processes to start up.

swap When there is insufficient room for all active processes on the machine to reside in main memory, the kernel begins removing some processes. The swap device is the temporary location where these processes spend time waiting to run again. Like the dump device, the swap device is auto-configured so the user need not worry about setting it. However, this can be overridden, and the swap device can be set to any unused portion of a minidisk.

swaplow Starting offset in the swap minidisk (in 4K blocks) where swapping occurs.

swapmap Maximum number of fragments of swap space.

swbufs Number of buffers for swapping.

timeslew: The time synchronization control parameter.

timezone Local time zone.

For this release, this value is always 1. Install/Maint changes the machine name if the user wishes. If the machine name differs from the default, **aixps**, Install/Maint places an entry in **/etc/system** to indicate that this value will be patched when the user rebuilds his kernel.

tokens Number of file access tokens.

tokloc Number of entries in token site request table.

users Estimate of number of users on site.

ulim_file The limit on the file size that a user may have. The units are in 512-byte blocks.

ulim_data The limit on the largest data segment size that a user may have. The units are in bytes.

ulim_stack
The limit on the largest stack segment size that a user may have. The units are in bytes.

vbuf The **vbuf** parameter is not used.

vsasz, pvsasz
These parameters are strictly for optimizing the number of dynamic allocation calls. **vsasz** is the number of **vsegs** allocated at one instance and **pvsasz** is the number of **pvsegs**. Increasing these values decreases the calls to the memory allocator but also decreases the amount of free memory available at a given instant. If **vseg** use is low, having high values would mean that memory utilization is being wasted. However, if **vseg** use is high, higher numbers could cause a performance increase.

vsegs **vsegs** (virtual segments) describe virtual objects such as user data, shared memory, executable text, and so on. Associated with each process are **pvsegs** (process virtual segments). **pvsegs** relates the **vseg** resources to a given process. Multiple processes may share the same **vseg** resource. The number of **vsegs** allocated reflects the number of separate system resource objects allowed. The value for this variable is based on the number of processes allowed for the system, so it does not need to change. The current formula is (number of processes * 4). Each **vseg** has a 104-byte overhead for the data structure.

vhandr Virtual hand rate. The frequency in ticks at which the page daemon wakes up and looks for memory to free. If this value is too high, the system spends too much time scanning the page

tables and performance is decreased. If it is too low, then the system won't be able to react quickly enough to fluctuations in system behavior, and again performance decreases.

vhandfrc Virtual hand fraction. Controls when the pager starts to run. It runs every **vhandr** ticks if free memory is greater than maximum memory divided by **vhandfrc**.

Example

The following sample system parameter stanza contains AIX Operating System information:

```
  locsite:  type = parm
            symbol = LOCSITE
            patchaddr = loc_site
            default = 0
```

File

/etc/master

Related Information

In this book: "mount" in topic 1.2.172, "attributes" in topic 2.3.5, "system" in topic 2.3.56, and "pty" in topic 2.5.21.

The **newkernel**, **osconfig** and **config** commands in *AIX Operating System Commands Reference*.

See "Generating a New Kernel" in *Managing the AIX Operating System*.

2.3.33 message

Purpose
Describes message, insert, and help formats.

Synopsis
 # include <msg10.h>

Description
The **puttext** command is used to convert message, text insert, and help descriptions from a format that can be edited into a format that can be accessed at run time. The descriptions in the file can be accessed by using the **msgimed, msgqued, msghelp,** and **msgrtrv** subroutines. The **gettext** command converts the descriptions back into a format that can be edited.

The file header contains a unique identifier indicating the type of file, a file format version number (currently 0), and the number of component entries in the file (currently, only one component entry per file is supported). The header file has the following form:

```
struct filehdr  {              /* FILE HEADER */
   char      unique[8];        /* unique file identifier "MSGSFILE" */
   unsigned short  version;    /* file format version number */
   unsigned short  numcomp;    /* number of component entries in file */
   };
```

Following the file header is the component index table. Each entry (currently, there is only one) in the table identifies the component, the national language (EN for English), the maximum index numbers that have been allocated and the offsets to the message index table, insert index table and help index table.

```
struct cmp_indx   {            /* Component index table entry */
  char           compid[6];    /* component ID */
  char           langid[2];    /* language ID */
  unsigned short flags;        /* reserved for flags (zero) */
  unsigned short maxnum[3];    /* max index numbers used for */
                               /* messages, inserts, and helps  */
  unsigned long offset[3];     /* offsets to msg, insert, and help */
                               /* index tables from start of file */
  unsigned long  reserved;     /* reserved */
  };
```

The component index table is followed by the message index table and message text, the insert index table and insert text, and help index table and help text. The header for each entry in the message, insert, and help index tables identifies the component ID and index number where the text actually resides, the offset to the text (and its length) if the text actually resides in this entry, the version number (used with a common file), and an indicator of whether the entry is current (can be accessed) or null.

```
             /* Format of header for entries in the    */
             /* message, insert, and help index tables */
             /* (Note that each index table must be    */
             /*  aligned on a long integer boundary.)  */

  #define MSGHEADR
     char           compid[6];  /* component ID for text source */
                                /* file ('======' or 'common')  */
```

```
        unsigned short  index;      /* index # for text source (zero   */
                                    /* indicates same index # */
        unsigned long   offset;     /* offset to text from start of   */
                                    /* index table                    */
        unsigned short  textlen;    /* text length (not incl null term) */
        unsigned short  version;    /* version  */
        unsigned short  flags;      /* flag definition */
                                    /*  01 off:  status = null */
                                    /*     on:  status = current */
                                    /*    (other flags reserved (zero))  */
        unsigned short reserve1;    /* reserved (zero) */

              /* flag definitions for MSGHEADR */
       #define mih_status 0x0001    /* off (0): status = null */
                                    /* on (1):  status = current */
```

Each entry in the insert index table contains only the header information.

```
      struct ins_indx   {           /* Insert table entry  */
                                    /* (contains header info. only)  */
         MSGHEADR                   /* header information */
         };
```

Each entry in the message index table and help index table contains the header information plus the title length (used for helps), a message/help manual reference, and the index number for the help associated with a message.

```
  struct mih_indx  { /* Entry in message or help index tables.  May */
                     /* also be used as entry in insert index table */
                     /* if only header information is referenced. */
     MSGHEADER                    /* header information */
     unsigned short titlelen;     /* title length (not incl null term ) */
     char           dcompid[3];   /* displayed component ID */
     char           dmsgid[3];    /* displayed message help ID */
     short          helpindx;     /* help index number (zero if no help, */
                                  /*  negative if help in common file) */
     unsigned short reserved;     /* reserved (zero)  */
  };
```

Each index table must be aligned on a long integer boundary.

Related Information

In this book: "msghelp" in topic 1.2.175, "msgimed" in topic 1.2.176, "msgqued" in topic 1.2.177, and "msgrtrv" in topic 1.2.179.

The **gettext** and **puttext** commands in *AIX Operating System Commands Reference*.

2.3.34 mh-alias

Purpose
Defines aliases for the Message Handling (MH) Package.

Description

The Message Handling (MH) Package supports both personal alias files and a system-wide alias file, **/usr/lib/mh/MailAliases**. Depending on the MH configuration, MH may also honor system-wide alias defined for the **sendmail** command. An alias file contains lines that associate an alias name with an address or a group of addresses.

Subtopics
2.3.34.1 File Format

2.3.34.1 File Format

Each line of an alias file has one of the following formats:

alias : address-group

alias ; address-group

<file

where

```
address-group   :=   address-list
                |    <file
                |    =AIX-group
                |    +AIX-group
                |    *

address-list    :=   address
                |    address-list, address
```

You can continue an alias definition on the next line by ending the line to be continued with the \ (backslash) character followed by the new-line character.

The **alias-file** and **file** parameters must be AIX file names. **AIX-group** must be a group name (or number) from **/etc/group**. **address** must be a simple Internet-style address. MH treats alias file names as case sensitive names but ignores case elsewhere in alias files.

If a line starts with the **<** (less than) character, MH reads the file specified after the **<** for more alias definitions. The reading is done recursively.

If an address group starts with the **<** (less than) character, MH reads the file specified after the **<** and adds the contents of that file to the address list for the alias.

If an address group starts with the **=** (equal) character, MH consults the **/etc/group** file for the AIX group specified after the equal character. MH adds each login name occurring as a member of the group to the address list for the alias.

If an address group starts with the **+** (plus) character, MH consults the **/etc/group** file to determine the group ID of the AIX group specified after the plus character. MH adds each login name occurring in the **/etc/passwd** file whose group ID is indicated by this group to the address list for the alias.

If an address group is specified as ***** (asterisk), MH consults the **/etc/passwd** file and adds all login names with a UID greater than 200, or the value set for everyone in **/usr/lib/mh/mtstailor** to the address list for the alias.

An approximation of the way the system resolves aliases at posting time:

1. The system builds a list of all addresses from the message to be delivered, eliminating duplicate addresses.

2. If the draft originated on the local host, the system performs alias

resolution for those addresses in the message that have no host specified.

3. For each line in the alias file, the system compares the alias with all existing addresses. If a match is found, the system removes the matched alias from the address list, and adds each new address in the address group to the address list if it is not already in the list. The alias itself is not usually output; the address group that the alias maps to is output instead. If the alias is terminated with a **;** (semicolon) instead of a **:** (colon), both the alias and the address are output in the correct form. (This correct form makes replies possible since MH aliases and personal aliases are unknown to the mail transport system.)

In the MH system, aliases in alias files are expanded into the headers of messages posted. This aliasing occurs first, at posting time, without the knowledge of the message transport system. In contrast, once the message transport system is given a message to deliver to a list of addresses, for each address that appears to be local, a system-wide alias file is consulted. These aliases are not expanded into the headers of messages delivered.

Since alias files are read line by line, forward references work, but backward references are not recognized. Although this forward referencing semantics prevents recursion, the **<alias-file** syntax may defeat this. Since the number of file descriptors is limited, such recursion will end when all file descriptors are depleted.

Example

The following example of a **mh-alias** file illustrates some features:

```
</user/lib/mh/DraftingAlias
temps:peggy,tina
tina:temp5@NODE3
p1:<project1.aliases
staff:=staff
support:+syssup
everyone:*
news.*:news
```

The first line says that more aliases should immediately be read from the file **/usr/lib/mh/DraftingAliases**. Following this, **tina** is defined as an alias for **temp5@NODE3**, and **temps** is defined as an alias for the two names **peggy** and **tina**. The definition of **p1** is given by reading the file *user_mh_directory*/**project1.aliases**. **staff** is defined as all users who are listed as members of the group **staff** in the **/etc/group** file, and **support** is defined as all users whose group ID in **/etc/passwd** is equivalent to the **syssup** group. Finally, **everyone** is defined as all users with a user ID in **/etc/passwd** greater than 200, and all aliases of the form **news.anything** are defined to be **news**.

Files

/usr/lib/mh/MailAliases The default system alias file.

/usr/lib/mh/mtstailor The MH tailor file.

Related Information
In this book: "group" in topic 2.3.26 and "passwd" in topic 2.3.44.

The **ali**, **conflict**, **post**, **send**, **sendmail**, and **whom** commands in *AIX Operating System Commands Reference*.

The "Overview of the Message Handling Package" in *Managing the AIX Operating System*.

2.3.35 mh-format

Purpose
Defines message formats for the Message Handling (MH) Package.

Description

Several MH commands use either a format string (similar to a **printf** string) or a format file to format their output. For example, the **scan** command uses a format string to format the scan listing for each message, while the **repl** command uses a format file to format a message reply.

A MH format string is similar to a **printf** string, but uses multi-letter escape sequences beginning with the **%** character. In addition, the usual C language backslash characters (\b, \f, \n, \r, and \t) are recognized. To continue a format line to the next line, precede the new-line character with a \ (backslash).

The interpretation model is based on a simple machine with two registers, **num** and **str**. The former contains an integer value; the latter a string value. When an escape is processed, if the escape requires an argument, the system reads the current value of either **num** or **str**, and if the escape returns a value, the system writes either **num** or **str**.

Note: The **MHL** command will look for inbox in $CWD/Mail instead of $HOME/Mail.

Subtopics
2.3.35.1 Escapes
2.3.35.2 mhl.format

2.3.35.1 Escapes

Escapes are of three types: component, function, or control. A component escape, specified as **%{name}**, is created for each header found in the message being processed. For example, **%{date}** refers to the **Date:** field of the appropriate message. A component escape is always string valued.

A control escape is one of:

```
%<escape
%|
%>
```

which corresponds to an if-then-else construct. If **escape** has a nonzero value (for integer-valued escapes) or is not empty (for string-valued escapes), then everything up to **%|** or **%>** (whichever comes first) is interpreted; otherwise, processing skips to **%|** or **%>** (whichever comes first) and starts interpreting again.

A function escape is specified as **%(name)** and is statically defined. Here is the list:

Escape	**Argument**	**Returns Interpretation**
nonzero	integer integer	**num** has a nonzero value
zero	integer integer	**num** is zero
eq	integer integer	**num** == width
ne	integer integer	**num** != width
gt	integer integer	width > **num**
null	string integer	**str** is empty
nonnull	string integer	**str** is not empty
putstr	string	Display **str**
putstrf	string	Display **str** in the specified width, for example: %20(putstrf{**subject**})
putnum	integer	Display **num**
putnumf	integer	Display **num** in the specified width, for example: %4(putnum(**msg**))
msg	integer	Message number
cur	integer	Message is current

size	integer	Size of message
strlen	string integer	Length of **str**
me	string	The user's mailbox
plus	integer	Add width to **num**
minus	integer	Subtract **num** from width
charleft	integer	Space left in output buffer
timenow	integer	Seconds from 00:00:00 GMT January 1, 1970

When **str** is a date, these escapes are useful:

Escape	**Argument**	**Returns Interpretation**
sec	string integer	Seconds of the minute
min	string integer	Minutes of the day
hour	string integer	Hours of the day (24-hour clock)
mday	string integer	Day of the month
mon	string integer	Month of the year
wday	string integer	Day of the week (Sunday=0)
year	string integer	Year of the century
yday	string integer	Day of the year
dst	string integer	Daylight savings in effect
zone	string integer	Time zone
sday	string integer	Day of the week known. Values are: **1** Explicit day **0** Implicit (the MH package figured it out) **-1** Unknown (the MH package could not figure it out)
clock	string integer	Seconds from 00:00:00 GMT January 1, 1970
rclock	string integer	Seconds prior to current time
month	string string	Month of the year
lmonth	string string	Month of the year (long form)

tzone	string string	Time zone
day	string string	Day of the week
weekday	string string	Day of the week (long)
tws	string string	RFC 822 rendering of the date
pretty	string string	A more user-friendly rendering
nodate	string	Date was not parseable

When **str** is an address, these escapes are useful:

Escape	**Argument**	**Returns Interpretation**
pers	string string	The personal name of the address
mbox	string string	The local part of the address
host	string string	The domain part of the address
path	string string	The route part of the address
type	string integer	The type of host. Values are: **-1** uucp **0** Local **1** Network **2** Unknown
nohost	string integer	No host was present in the address
ingrp	string integer	The address appeared inside a group
gname	string string	The name of the group (present for first address only)
note	string string	Commentary text
proper	string string	RFC 822 rendering of the address
friendly	string string	A more user-friendly rendering
mymbox	string	The address refers to the user's mailbox
formataddr	string	Display **str** in an address list

2.3.35.2 mhl.format

The **mhl.format** file is similar to other MH format files, but controls the format of output when **mhl** is the message listing program. Each line of the **mhl.format** file must have one of the following forms:

```
;comment
:cleartext
variable[,variable...]
component:[variable...]
```

A line beginning with the **;** character contains comments that are ignored. A line beginning with the **:** character contains text for output. A line that contains the **:** character only produces a blank output line. A line beginning with **component** defines the format of the specified component. If a variable follows a component, the variable applies only to that component. Lines having other formats define the global environment.

Variables that have integer or string values as arguments must be followed by an **=** (equal) character and the integer or string value (for example, **overflowoffset=5**). String values must also be enclosed in double quotation characters (for example, **overflow"***"**). An argument specified with the suffix **/G** has global scope. An argument specified with the suffix **/L** has local scope.

The entire **mhl.format** file is parsed before output processing begins. Thus, if a variable's global setting is defined in multiple places, the last global definition for that variable describes the real global setting.

The following table lists the **mhl.format** variables and their arguments:

Variable	**Argument**	**Description**
width	integer	Set the screen width or component width
length	integer	Set the screen length or component length
offset	integer	Indent **component** the specified number of columns
overflowtext	string	Output **string** at the beginning of each overflow line
overflowoffset	integer	Indent overflow lines the specified number of columns
compwidth	integer	Indent component text the specified number of columns after the first line of output
uppercase	flag	Output text of **component** in all upper case characters
nouppercase	flag	Output text of **component** in the case entered

clearscreen	flag/G	Clear the screen before each page
noclearscreen	flag/G	Do not clear the screen before each page
bell	flag/G	Produce an audible indicator at the end of each page
nobell	flag/G	Do not produce an audible indicator at the end of each page
component	string/L	Use **string** as the name for the specified **component** instead of the string **component**
nocomponent	flag	Do not output the string **component** for the specified **component**
center	flag	Center **component** on line. This variable works for one-line components only.
nocenter	flag	Do not center **component**
leftadjust	flag	Strip off the leading whitespace characters from each line of text
noleftadjust	flag	Do not strip off the leading whitespace characters from each line of text
compress	flag	Change new-line characters in text to space characters
nocompress	flag	Do not change new-line characters in text to space characters
formatfield	string	Use **string** as the format string for the specified **component**
addrfield	flag	The specified **component** contains addresses
datefield	flag	The specified **component** contains dates.
ignore	unquoted string	Do not output component specified by **string**

Example

The following format string is the default for the **scan** command. It has been divided into several pieces for readability. The first part is:

```
  %4(putnum(msg))%<(cur)+%| %>%<{replied}-%| %>
```

This says that the message number should be displayed in four digits. If the message is the current message, a + character is displayed next; otherwise, a space character is displayed. If a **replied:** field is

present, then a - (minus) is displayed; otherwise, a space is displayed. Next:

```
%02(putnumf(mon{date}))%02(putnumf(mday{date}))
```

The month and day are displayed in two digits (zero filled). Next:

```
%<{date} %|*>
```

If no **date:** field is present, then an ***** (asterisk) character is displayed; otherwise, a space character is displayed. Next:

```
%<(mymbox{from})To:%14(putstr(friendly{to}))
```

If the message is from me, display **to:** followed by a friendly rendering of the first address in the **To:** field. Continuing:

```
%|%17(putstrf(friendly{from}))%
```

If the message is not from me, display **from:** followed by the **from:** address. And finally:

```
%{subject}<<%{body}>>
```

Display the subject and initial body of the message.

This method of formatting messages allows you to extract individual fields and display them in the format you desire.

If you use the **-form file** argument when you run **scan**, it treats each line in **file** as a format string and acts accordingly. The following files contain **scan** listing formats that you can look at: **/usr/lib/mh/scan.time**, **/usr/lib/mh/scan.size**, and **/usr/lib/mh/scan.timely**.

The following line is an example of a line that could appear in the **mhl.format** file:

```
width=80,length=40,clearscreen,overflow"***",overflowoffset=5
```

This format line defines the screen size to be **80** columns by **40** rows and specifies that the screen should be cleared before each page, that the overflow text should be flagged with the string *******, and that the overflow indentation should be **5** columns.

Related Information

The **ap**, **dp**, **mhl**, and **scan** commands in *AIX Operating System Commands Reference*.

The "Overview of the Message Handling Package" in *Managing the AIX Operating System*.

2.3.36 mh-mail

Purpose
The message format for the Message Handling (MH) Package.

Description
The Message Handling (MH) Package processes messages in a particular format. While this format is different from that used by the Bell and Berkeley mail systems, the MH package can read Bell and Berkeley message files.

Each user has a mail drop that initially receives all messages processed by the **post** or **spost** command. The **inc** command reads from that mail drop and incorporates the new messages found there into the user's own mail folder (typically **inbox**). The mail drop consists of one or more messages.

Messages are expected to consist of lines of text. Graphics and binary data are not handled. No data compression is accepted. All text is ASCII 7-bit data.

The general memo framework of the ARPA Internet RFC 822 standard is used. A message consists of a block of information in a rigid format, followed by general text with no specified format. The rigidly formatted first part of a message is called the header, and the free-format portion is called the body. The header must always exist, but the body is optional. These parts are separated by an empty line, that is, two consecutive new-line characters. Within messages, the header and body may be separated by a line consisting of dashes:

```
To:
cc:
Subject
-------
```

The header is composed of one or more header components. Each header component can be viewed as a single logical line of ASCII characters. If the text of a header component extends across several lines, the continuation lines are indicated by leading spaces or tab characters.

Each header component is composed of a keyword or name, along with associated text. The keyword begins at the left margin, may not contain space or tab characters, may not exceed 63 characters, and ends with a **:** (colon). Certain components (as identified by their keywords) must follow rigidly the defined formats in their text portions.

The text for most formatted components (for example, **Date:** and **Message-Id:**) is produced automatically. The text for other components must be entered by the user (for example, **To:** and **cc:**). Multiple addresses are separated by commas. A missing host/domain is assumed to be the local host domain.

Header Components

Date: Added by **post**, **spost**, or the mail transport system; contains the date and time of the message's entry into the transport system.

From: Added by **post** or **spost**; contains the address of the author or authors (may be more than one if a **Sender** field is present). Replies are typically directed to

addresses in the **Reply-To:** or **From:** field (the former has precedence if both are present).

Sender: Added by **post** or **spost** in the event that the message already has a **From:** line. This line contains the address of the actual sender. Replies are never sent to addresses in the **Sender:** field.

To: Contains addresses of primary recipients.

cc: Contains addresses of secondary recipients

Bcc: Contains still more recipients. However, the **Bcc:** line is not copied into the message as delivered, so these recipients are not listed. The MH package uses an encapsulation method for blind copies.

Fcc: Causes the **post** or **spost** command to copy the message into the specified folder for the sender, if the message was successfully given to the transport system.

Message-Id: A unique message identifier added by **post** or **spost** if the **-msgid** flag is set.

Subject: Sender's commentary. It is displayed by the **scan** command.

In-Reply-To: A commentary line added by the **repl** command when replying to a message.

Resent-Date: Added when redistributing a message by **post** or **spost**.

Resent-From: Added when redistributing a message by **post** or **spost**.

Resent-To: New recipients for a message resent by the **dist** command.

Resent-cc: More recipients. See **cc:** and **Resent-To:**.

Resent-Bcc: More recipients. See **Bcc:** and **Resent-To:**.

Resent-Fcc: Copies resent message into a folder. See **Fcc:** and **Resent-To:**.

Resent-Message-Id: A unique identifier attached by **post** or **spost** if you specify the **-msgid** flag. See **Message-Id:** and **Resent-To:**.

Resent: Annotation that **dist** uses when you specify the **-annotate** flag.

Forwarded: Annotation that the **forw** command uses when you specify the **-annotate** flag.

Replied: Annotation that **repl** uses when you specify the **-annotate** flag.

File

$HOME/.newmail Location of mail drop

Related Information
Standard for the Format of ARPA Internet Text Messages, RFC 822. See "Related Network Publications" in topic 1.2.277.4.

The "Overview of the Message Handling Package" in *Managing the AIX Operating System*.

2.3.37 mhook

Purpose

Specifies actions to be taken when mail is received.

Description

An ***mhook*** (or receive-mail hook) is an action that is automatically performed when new mail is received through the Message Handling (MH) Package. Whenever you receive a new message, the **sendmail** command searches for the file **.forward** in your **$HOME** directory. **sendmail** pipes the new message to the **slocal** program when **$HOME/.forward** exists and contains the following line:

```
| /usr/lib/mh/slocal
```

The **slocal** program reads the file **$HOME/.maildelivery** and performs the actions specified in that file for each message being delivered. You can specify your own mail delivery instructions (or **mhooks**) in **$HOME/.maildelivery**. Each line in **$HOME/.maildelivery** describes an action and the conditions under which the action should be performed. Each line must contain five arguments separated by commas or space characters. These arguments are:

field pattern action result string

The following list describes each argument:

field Specifies a header component field to be searched for a match to the character string specified in the **pattern** argument. You can specify one of the following values for the **field** argument:

component Searches the specified header component.

***** Always matches.

addr Searches whatever field was used to deliver the message to you.

default Matches only if the message has not been delivered yet.

source Specifies the out-of-band sender information.

pattern Specifies the character string to search for in the header component given by the **field** argument. The **pattern** argument is not case sensitive. Thus, the character string matches any combination of upper case and lower case characters. You must specify a dummy pattern if you use ***** or **default** in the **field** argument.

action Specifies an action to take with the message if the message contains the pattern specified in the **pattern** argument. You can specify the following actions:

file or **>** Appends the message to the file given by the **string** argument. If the message can be written to the file, the action is considered successful. When a message is appended to a file, the header component

Delivery-Date: is added to the message to indicate when the message was appended to the file.

pipe or **|** Pipes the message as standard input to the command named by the **string** argument, using the shell to interpret the string. If the exit status from the command is 0 (zero), the action is considered successful. Prior to giving the string to the shell, the string is expanded with the following built-in variables:

$(sender) The return address for the message.

$(address) The address that was used to deliver the message.

$(size) The size of the message in bytes.

$(reply-to) Either the **Reply-To:** or **From:** header component of the message.

$(info) Miscellaneous out-of-band information.

qpipe or **^** Similar to **pipe**, but executes the command directly after built-in variable expansion without assistance from the shell. If the exit status from the command is 0 (zero), the action is successful.

destroy Always succeeds.

result Indicates how the action should be performed. You can specify one of the following values for this argument:

A Performed the action. If the action succeeds, the message is considered delivered.

R Performs the action. Even if the action succeeds, the message is not considered delivered.

? Performs the action only if the message has not been delivered. If the action succeeds, the message is considered delivered.

string If you use **file** as the **action** argument, **string** specifies the file to which the message can be appended. If you use **pipe** or **qpipe**, **string** specifies the command to execute. If you use **destroy** as the **action** argument, **string** is not used, but you must still include a dummy **string** argument.

All five arguments must be present in each line of the file. Blank lines in **.maildelivery** are ignored. Put a # in the first column to indicate a comment.

If **.maildelivery** cannot be found, or does not deliver the message, **/usr/lib/mh/maildelivery** is used in the same manner. If the message is still not delivered, it is delivered to the user's maildrop, **$HOME/.newmail**.

MH contains four standard programs that can be run as receive-mail hooks: **rcvdist**, **rcvpack**, **rcvstore**, and **rcvtty**. *AIX Operating System Commands*

Reference contains descriptions of these programs.

Example

The following example shows some lines that can be specified as **mhooks** in **$HOME/.maildelivery**:

```
# If the message is from George, save it.
From    george          file    A       george.mail

# If the message is to the project manager, save a copy in log.
addr    manager         >       R       proj_X/statlog
# and forward it to Amy
addr    manager         |       A       "/usr/lib/mh/rcvdist amy"

# Save any messages not delivered.
default   -             >       ?       mailbox
```

Files

$HOME/.forward	The file searched by the **sendmail** command when mail is received.
/usr/lib/mh/slocal	The **slocal** program that, when specified in **$HOME/.forward**, reads the file **$HOME/.maildelivery** and performs the actions specified in that file for each message being processed.
/usr/lib/mh/maildelivery	The mail delivery instructions. **$HOME/.forward**, performs actions defined in **$HOME/.maildelivery**.
$HOME/.maildelivery	The file specifying receive-mail hooks for **slocal** to perform.

Related Information

The **rcvdist**, **rcvpack**, **rcvstore**, **rcvtty**, **sendmail**, and **slocal** commands in *AIX Operating System Commands Reference*.

The "Overview of the Message Handling Package" in *Managing the AIX Operating System*.

2.3.38 mh-profile

Purpose

Customizes the Message Handling (MH) Package.

Description

Each user of the Message Handling (MH) Package is expected to have a file named **.mh_profile** in the home directory. This file contains a set of user parameters used by some or all of the MH programs. Each line of the file is in the following format:

profile-entry: value

Profile Entries

Of the possible profile entries, only **Path:** is required. The others are optional. Some entries have default values if the entries are not present. In the notation used in the following list (profile, default) indicates whether the information is kept in the user's MH profile or context file and indicates the default value.

Path: Specifies the location of the *user_mh_directory* directory. The usual location is **$HOME/Mail**. (profile, no default)

context: Declares the location of the MH context file. (profile, default: *user_mh_directory*/**context**)

Current-Folder: Keeps track of the current open folder. (context, default: **inbox**)

Previous-Sequence: Names the sequences that should be defined as the **msgs** or **msg** argument given to the program. If not present, or empty, no sequences are defined. Otherwise, for each name given, the sequence is first set to zero and then each message is added to the sequence. (profile, no default)

Sequence-Negation: Defines the string which, when prefixed to a sequence name, negates that sequence. Thus, if **Sequence-Negation:** is set to **not**, **notseen** means all those messages that are not a member of the sequence **seen**. (profile, no default)

Unseen-Sequence: Names the sequences that are defined as those messages recently incorporated by the **inc** command. The **show** command removes messages from this sequence after they have been seen. If not present, or empty, no sequences are defined. Otherwise, for each name given, the sequence is first set to zero and then each message is added to the sequence. (profile, no default)

mh-sequences: Names the file in each folder that defines public sequences. To disable the use of public sequences, leave the value of this entry blank. (profile, default: **.mh_sequences**)

atr-seq-folder: Keeps track of the private sequence named **seq** in the specified **folder**. (context, no default)

Editor: Defines the editor to be used by the **comp**, **dist**, **forw**, and **repl** commands. (profile, default: **prompter**)

Msg-Protect: Defines octal protection bits for message files. See the **chmod** command in *AIX Operating System Commands Reference* for an explanation of the octal number. (profile, default: **0644**)

Folder-Protect: Defines protection bits for folder directories. (profile, default: **0711**)

program: Sets default flags to be used whenever the specified MH **program** is invoked. For example, you can override the **Editor:** profile component when replying to messages by adding the profile entry:

```
repl: -editor /bin/ed
```

(profile, no defaults)

lasteditor-next: Specifies the editor that is the default editor after using **lasteditor**. This takes effect at the **What now?** level of the **comp**, **dist**, **forw**, and **repl** commands. After editing the draft with **lasteditor**, the default editor is set to be **nexteditor**. If you enter **edit** without any arguments to **What now?**, then **nexteditor** is used. (profile, no default)

Folder-Stack: Defines the contents of the folder stack of the **folder** command. (context, no default)

Alternate-Mailboxes: Tells the **repl** and **scan** commands which addresses are really yours. In this way, **repl** knows which addresses should be included in the reply, and **scan** knows if the message really originated from you. Addresses must be separated by a comma, and the host names listed should be the official host names for the mailboxes you indicate, as local nicknames for hosts are not replaced with their official site names. For each address, if a host is not given, then that address on any host is considered to be you. In addition, an asterisk may appear at either or both ends of the mailbox and host to indicate wildcard matching. (profile, default: **$LOGNAME**)

Draft-Folder: Indicates a default draft folder for the **comp**, **dist**, **forw**, and **repl** commands. (profile, no default)

digest-issue-list: Tells **forw** the last issue of the last volume sent for the digest **list**. (context, no default)

digest-volume-list: Tells **forw** the last volume sent for the digest **list**. (context, no default)

MailDrop: Tells **inc** your mail drop, if different from the

default. This is superseded by the **$MAILDROP** environment variable. (profile, default: **$HOME/.newmail**)

Signature: Tells **inc** your mail signature. This is superseded by the **$SIGNATURE** environment variable. (profile, no default)

The following profile elements are used whenever a MH program invokes some other program. You can use **.mh_profile** to select alternate programs. The following list gives the default values.

fileproc: /usr/bin/refile

incproc: /usr/bin/inc

installproc: /usr/lib/mh/install-mh

lproc: /bin/pg

mailproc: /usr/bin/mhmail

mhlproc: /usr/lib/mh/mhl

moreproc /bin/pg

mshproc: /usr/bin/msh

packproc: /usr/bin/packf

postproc: /usr/lib/mh/spost (1)

rmmproc: none

rmfproc: /usr/bin/rmf

sendproc: /usr/bin/send

showproc: /bin/pg

whatnowproc: /usr/bin/whatnow

whomproc: /usr/bin/whom

When you invoke a MH program, it reads the **.mh_profile** file by default. If you define the environment variable **$MH**, you can specify another profile file. If the file of **$MH** is not absolute (does not begin with /), it is presumed to start in the current directory. This is one of the few exceptions in the MH package where nonabsolute path names are not considered relative to your MH directory.

Similarly, if you define the environment variable **$MHCONTEXT**, you can specify a context other than the normal context file (as specified in the MH profile). If the value of **$MHCONTEXT** is not absolute, it is presumed to start from your MH directory.

MH programs also support the following other environment variables:

$MAILDROP Tells **inc** the default mail drop. This supersedes the **MailDrop:** profile entry.

$SIGNATURE Tells **send** and **post** your mail signature. This supersedes the **Signature:** profile entry.

$HOME Tells all MH programs your home directory.

$TERM Tells the MH package your terminal type. The **TERMCAP** variable is also consulted. In particular, these tell **scan** and **mhl** how to clear your terminal and how many columns wide your terminal is. They also tell **mhl** how many lines long your terminal screen is.

$editalt Specifies an alternate message. This is set by **dist** and **repl** during edit sessions so that you can read the message being distributed or replied to. This message is also available through a link called @ in the current directory if your current directory and the folder the message lives in are on the same AIX file system.

$mhdraft Specifies the path of the working draft.

$mhfolder Specifies the folder containing the alternate message. This is set by **dist** and **repl** during edit sessions so you can read other messages in the current folder besides the one being distributed. The **$mhfolder** environment variable is also set by **show**, **prev**, and **next** for use by **mhl**.

Files

$HOME/.mh_profile The user profile.

user_mh_directory/**context** The user context file.

folder/.mh_sequences Public sequences for **folder**.

Related Information
In this book: "environment" in topic 2.4.6.

The **mh** command in *AIX Operating System Commands Reference*.

The "Overview of the Message Handling Package" in *Managing the AIX Operating System*.

(1) The **spost** command uses the address handling capabilities of the **sendmail** command. If you do not have **sendmail** installed on your system, set the **postproc:** profile entry to **/usr/lib/post**.

2.3.39 mh-tailor

Purpose
Defines how MH commands work.

Description

The entries located in the file **/usr/lib/mh/mtstailor** tailor how MH commands work.

File Entries

The following list describes the **/usr/lib/mh/mtstailor** file entries and their default values. All of the file entries are optional.

File Entry	Description
localname:	Specifies the host name of the local system. If this entry is not defined, MH queries the system for the default value.
systemname:	Specifies the host name of the local system in the uucp domain. If this entry is not defined, MH queries the system for the default value.
mmdfldir:	Specifies the location of mail drops. If this entry is not present, or is present and empty, mail drops are located in the user's **$HOME** directory.
mmdflfil:	Specifies the name of the file used as the maildrop. If this entry is not defined, the default file name is **.newmail**.
mmdelim1:	Specifies the beginning-of-message delimiter for mail drops. The default value is **\001\001\001\001\n**.
mmdelim2:	Specifies the end-of-message delimiter for mail drops. The default value is **\001\001\001\001\n**.
mmailid:	Specifies whether support for MMailids in **/etc/passwd** is enabled. If **mmailid:** is set to a nonzero value, support is enabled. The **pw_gecos** field in the password file has the form: **My Full Name mailid** When support for MMailids is enabled, the internal MH routines that deal with user and full names return **mailid** and **My Full Name** respectively. The default value is 0 (zero).
lockstyle:	Specifies the locking discipline. The default value is 0.
lockldir:	Specifies the directory for locked files. This entry is not used when **lockstyle** is 0.
sendmail:	Specifies the path name of the **sendmail** program. The default value is **/usr/lib/sendmail**.
maildelivery:	Specifies the path name of the file containing the system

default maildelivery instructions. The default value is **/usr/lib/mh/maildelivery**.

everyone: Specifies the users to receive messages addressed to everyone. All users having UIDs greater than the specified number (not inclusive) receive messages addressed to everyone. The default value is 200.

File

/usr/lib/mh/mtstailor The MH tailor file.

Related Information

The **ap**, **conflict**, **inc**, **msgchk**, **msh**, **post**, **rcvdist**, and **rcvpack** commands in *AIX Operating System Commands Reference*.

The "Overview of the Message Handling Package" in *Managing the AIX Operating System*.

2.3.40 mntent, mtab

Purpose
Contains information about mounted file systems.

Synopsis

#include <mntent.h>

Description

The file **/etc/mtab** describes the file systems currently mounted on the local machine. It is a symbolic link to a local-only file. To find out about file systems mounted on other sites in a TCF cluster, specify a specific site's **<LOCAL>/mtab** file to the **setmtnent** routine.

The file **/etc/mtab** is created at system-startup time to include the root and **<LOCAL>** file systems by the **setmnt** command. When run, the **mount** command adds entries to the **/etc/mtab** file, and the **umount** command removes entries.

The **/etc/mtab** file consists of a number of lines of the form:

```
fsname dir type opts freq passno flags gfs pack time
```

For example,

```
/dev/xy0a/ufs rw,noquota 1 2 40 1 7 604161472
```

The entries in this file are accessed using the routines in **getmntent**, which returns a structure of the following form:

```
struct mntent {
        char    *mnt_fsname;   /* file system name */
        char    *mnt_dir;      /* file system path prefix */
        char    *mnt_type;     /* ufs, nfs */
        char    *mnt_opts;     /* ro, quota, etc. */
        int     mnt_freq;      /* dump frequency, in days */
        int     mnt_checkno;   /* check number for parallel fsck */
        char    *mnt_flags;    /* file system flags */
        gfs_t   mnt_gfs;       /* global file system numbers */
        pckno_t mnt_pack;      /* pack number */
        long    mnt_time;      /* time when mounted */
};
```

There is one entry per line in the file and the fields are separated by blanks.

The **mnt_opts** field consists of a string of options separated by commas. Some of the options are common to all file system types while others only make sense for a single file system type. For more information on the options available with the **mount** command, see *AIX Operating System Commands Reference*.

The **mnt_type** field determines how the **mnt_fsname** and **mnt_opts** fields will be interpreted. Below is a list of file system types currently supported and the way in which each interprets these fields:

ufs	**mnt_fsname**	Must be a block special device.

	mnt_opts	Valid options are **ro**, **rw**, **suid**, **nosuid**, **quota**, **noquota.**
nfs	**mnt_fsname**	The path on the server of the directory to be served.
	mnt_opts	Valid options are **ro**, **rw**, **suid**, **nosuid**, **hard**, **soft**, **bg**, **fg**, **retry**, **rsize**, **wsize**, **timeo**, **retrans**, **port**, **intr**.

The **mnt_freq** field indicates how often each partition should be dumped by the **dumpbsd** command, and also triggers the **w** option of the **dumpbsd** command indicating the file systems that should be dumped. Most systems set the **mnt_freq** field to 1, indicating that the file systems are dumped each day.

The **mnt_checkno** field is used by the disk consistency check program **fsck** to control simultaneous checking of file systems during a reboot. To avoid having two file systems checked simultaneously, make sure the values of **mnt_checkno** are the same.

The **mnt_flags** field corresponds to the **s_flags** field in the file system super block. For more information, see "fs" in topic 2.3.20.

The **mnt_gfs** field indicates the global file system number for the file system.

The **mnt_pack** field indicates the global file system pack number for the file system.

The **mnt_time** field indicates the time the file system was mounted by specifying the number of seconds since Jan 1, 1970 00:00:00 GMT.

File
/etc/mtab

Related Information
In this book: "getmntent, setmntent, addmntent, endmntent, hasmntopt" in topic 1.2.104.

The **fsck**, **mount**, **quotacheck**, and **quotaon** commands in *AIX Operating System Commands Reference*.

2.3.41 netparams

Purpose

Describes the TCF (Transparent Computing Facility) cluster networking parameters.

Description

The **/etc/netparams** file contains information on the TCF cluster networking parameters. Lines in the file, starting with "#" are comment lines and are discarded. The fields are:

site Indicates the site number of the TCP cluster site. Valid ranges for this field are 1 through 31.

timeout Indicates the number of "network ticks" between retransmissions if no acknowledgement is received. (At present, a network tick corresponds to 2/3 of a second.) Valid ranges for this field are 1 through 255.

retries Specifies the number of retransmissions a TCF network message will be attempted before TCF site is considered to be down. If the TCF site was known to be up just prior to sending this message, then the number of **retries** is multiplied by 3 (for example, if the **retries** field specifies "3" and if the site was not up before, the message is sent 4 times--one original message plus 3 retransmissions; if the site was up before, then the message would be sent 10 times--one original message plus 3*3 transmissions). Valid ranges for this field are 1 through 255.

window Specifies the number of "normal" (non-topology change) messages that may be sent by one site to another without an intervening acknowledgement, before the transmitting site will wait for a receipt of an acknowledgement. The windowing algorithm is "fixed" (non-dynamic) in nature. Valid ranges for this field are 2 through 255.

special Specifies the number of "special" (topology change) messages that may be sent without an intervening acknowledgement. These two windows act independently of one another. Valid ranges for this field are 2 through 255.

checksum Indicates whether or not checksums should be calculated and inserted into TCF messages that are being transmitted to the TCF site in question. The receiving site only checks for checksum correctness if the check field(s) are non-zero. Valid checksum fields are "yes" or "no" (case insensitive). A valid checksum is guaranteed to be non-zero.

Related Information
In this book: "netctrl" in topic 1.2.185.

The **netparams**, **inetstat**, and **probe** commands in the *AIX Operating System Command Reference*.

2.3.42 openfiles

Purpose
Specifies files to be kept open by **cron**.

Description
For performance reasons, if you want your most heavily used directories and files to be kept open by the **cron** command, you must specify their pathnames in the **openfiles** special file.

A default **openfiles** is provided with your system. When you edit this file, you may include comments preceded by a pound sign (#) in the first column position; any blank lines in the file are ignored by **cron**.

The maximum number of pathnames you may specify in **openfiles** is 48.

File

/etc/openfiles

Related Information
The **cron** command in the *AIX Operating System Commands Reference*.

2.3.43 options

Purpose
Defines the valid choices for each **ddi** option.

Description
The **/etc/ddi/options** file contains a sorted list of the valid choices for each keyword used in **ddi** files. The **devices** command uses this file to display the valid choices for the keywords during the **add**, **change**, and **showdev** subcommands.

Each line must follow the following format:

optionchoices

where:

option This field is exactly 20 characters long, is padded on the right with spaces, and contains no tab characters. An **option** is one of the following:

keyword	The keyword for which the valid choices are to be specified
keywordadapter	The keyword followed by the adapter name
keywordclass	The keyword followed by the device class
keywordtype	The keyword followed by the device type
keywordstanza	The keyword followed by the name of the device stanza in the **system** file.

The **devices** command looks for one of these combinations based on the setting of the **opts** keyword in the **kaf** file for the device.

choices This field is exactly 29 characters long, is padded on the right with spaces, and contains no tab characters.

Note: The **/etc/ddi/options** file must be sorted alphabetically by the **option** field. If it is not sorted, then the **devices** command displays incorrect information about the options available for a given keyword.

The use of extended characters in the **/etc/ddi/options** file is not supported.

File

/etc/ddi/options

Related Information
In this book: "ddi" in topic 2.3.13, "descriptions" in topic 2.3.14, and "kaf" in topic 2.3.30.

2.3.44 passwd

Purpose
Contains passwords and user account information.

Synopsis

#include <pwd.h>

Description
The **passwd** file is an ASCII file that contains all the information that defines a user on the system. It contains the following information:

Login nam
Encrypted passwor
Numerical user I
Numerical group I
Additional data for each use
Initial current director
Program to use as shell

Each field is separated from the next by a colon. The file has general read permission and the passwords are encrypted. Therefore, a user can use the file to map numerical user IDs to names without potentially compromising the security of other users.

The **adduser** command is used to maintain this file. Programs should use the **getpwent** subroutines to extract various fields in this file.

If the user password field is null, the user has no password. If the program field is null, the shell (**/bin/sh**) is used. The program field can contain parameters passed when the **exec** system call is issued. Parameters are separated by space (such as a space or tab characters). A \ (backslash) is used for escapement when a parameter contains a space. The **login** command accepts the program name and as many as 14 parameters. Any more than 14 parameters are ignored. A maximum of 4096 characters can be used for the program name and its parameters. More than 4096 characters causes **login** to exit. Parameters in this field can use symbolic escapement for the following special characters: \n, \r, \v (produces 013), \b, \t, and \f. Additionally, \0 through \7 builds a one-byte octal number. Anything else that is preceded by a \ (backslash) passes through.

The contents of the additional data for each user has the following format:

full_name / file_limit ; site_info;site_exec_perm

where:

full_name Contains the name of the user whose 8-character (or fewer) login name is in the first field.

If a user group list is used, the list may not exceed 500 eight-character login names.

file_limit Specifies the maximum length file the user can create. The length is specified as the number of 512-byte blocks. See the **login** command in *AIX Operating System Commands Reference* and the **ulimit** system call.

site_info Contains any printable character other than a colon or semicolon. This subfield is unused by the system software and is available for information for each user as required by applications specific to the site.

site_exec_perm Contains a site group name identifying the sites on which this user can log in and/or execute programs. (See "sitegroup" in topic 2.3.55 to find the site group name.)

Any or all of the subfields can be omitted. If the **file_limit** subfield is omitted, the preceding **/** (slash) is omitted and the system-wide default limit is used. If the **site_info** and **site_exec_perm** subfields are omitted, the **;** (semicolon) preceding each of these subfields must also be omitted. If, however, the **site_info** subfield is omitted but the **site_exec_perm** subfield is present, the **;** preceding each of these subfields must be present.

Subtopics
2.3.44.1 Passwords

2.3.44.1 Passwords

The encrypted password is 13 characters long. The characters used come from the extended characters (code page P0, see "data stream" in topic 2.4.3) and may be uppercase or lowercase characters, numerals, and the . (period) and / (slash) characters except when the password is null. In this case, the encrypted password is also null. Password aging affects a particular user if a comma and a string of characters that are not null follows the user password in this file. Such a string must be initially introduced by a person with superuser authority.

The first character of the age, **M** for example, is the maximum number of weeks a password is valid. The next character, **m** for example, is the minimum number of weeks, before the password can be changed. The remaining characters indicate when the password was last changed, given as the number of weeks since the beginning of 1970 to the time of the password change. A null string is equivalent to 0. **M** and **m** have numerical values in the range 0 through 63. If **m = M = 0**, the user is forced to change the password at the next login. This causes the age to disappear from the password file entry. If **m > M**, only someone with superuser authority is able to change the password.

Note: All userid's and passwords must contain ASCII characters only.

File

/etc/passwd

Related Information

In this book: "a64l, l64a" in topic 1.2.6, "crypt, encrypt, setkey" in topic 1.2.52, "getpwent, getpwuid, getpwnam, setpwent, endpwent" in topic 1.2.114, "ulimit" in topic 1.2.313, "group" in topic 2.3.26, "sitegroup" in topic 2.3.55, and "data stream" in topic 2.4.3.

The **login** and **passwd** commands in *AIX Operating System Commands Reference*.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

2.3.45 plot

Purpose
Provides the graphics interface.

Description
The **tplot** commands interpret these graphics files for various devices, performing the plotting instructions in the order that they appear.

A graphics file consists of a stream of plotting instructions. Each instruction consists of an ASCII letter usually followed by bytes of binary information. A point is designated by 4 bytes representing the **x** and **y** values; each value is a two-byte signed integer. The last designated point in an **l**, **m**, **n**, or **p** instruction becomes the current point for the next instruction.

The following table lists each of the **plot** instructions and the corresponding **plot** subroutines.

Instr	Sub	Description
a	**arc**	Draws the arc described by the following 12 bytes. The first 4 bytes describe the center point (x, y) of the arc or circle. The second 4 bytes describe the beginning point of the arc. The third 4 bytes describe the ending point of the arc. Arcs are drawn counterclockwise. The results are unpredictable if the three points do not really form an arc.
c	**circle**	Draws a circle whose center point is defined by the first 4 bytes, and whose radius is given as an integer in the following two bytes.
e	**erase**	Starts another frame of output.
f	**linemod**	Uses the following string, terminated by a new-line character, as the style for drawing further lines. The styles are dotted, solid, long-dashed, short-dashed, and dot-dashed.
l	**line**	Draws a line from the point designated by the next 4 bytes to the point designated by the following 4 bytes.
m	**move**	The next 4 bytes designate a new current point.
n	**cont**	Draws a line from the current point to the point designated by the next 4 bytes.
p	**point**	Plots the point designated by the next 4 bytes.
s	**space**	The next 4 bytes designate the lower left corner of the plotting area; followed by 4 bytes for the upper right corner. The plot is magnified or reduced to fit the device as closely as possible.
t	**label**	Places the following ASCII string so that its first character falls on the current point. A new-line character terminates the string.

The space setting

```
  space(0, 0, 480, 432);
```

exactly fills the plotting area with unity scaling for the IBM Personal Computer Graphics Printer. The upper limit is immediately outside the plotting area, which is taken to be square. Points outside the plotting area can be displayed on devices that do not have square displays.

Related Information
In this book: "plot" in topic 1.2.206 and "TERM" in topic 2.4.26.

The **graph** and **tplot** commands in *AIX Operating System Commands Reference*.

2.3.46 ports

Purpose
Describes the ports.

Description
The **ports** file contains the names and characteristics of all the system terminal ports. It provides a convenient means to associate values with named keyword parameters on a port-by-port basis, with defaults supplied as desired.

The **getty** process is the principal user of the information in this file. Since programs using this file look for specific keyword parameters and ignore all others, parameters other than those discussed here can be added to this file as necessary.

Subtopics
2.3.46.1 File Format
2.3.46.2 Port-Control Parameters
2.3.46.3 Other Port Parameters

2.3.46.1 File Format

The **ports** file consists of one or more named stanzas usually separated by blank lines. Each stanza begins with its name followed by a colon, and contains assignments of values to keyword attributes. The values, in turn, may be alphanumeric strings or arbitrary character strings enclosed in double quotes.

Stanzas headed by the name **default** specify attribute-value pairs that are associated with all of the ports following it to the next **default** stanza. Explicit values within a **port** stanza override this association.

2.3.46.2 Port-Control Parameters

Most of the parameters in the **ports** file are port controls for login terminals. Because there are system defaults, specified in the **getty** process, it is not usually necessary to specify more than a few attributes in the **ports** file, as in the example. The port control parameters and their meanings are as follows:

aa When the value of **aa** is TRUE, the following Hayes modem command sequences are output:

- **+++** Hayes command escape sequence.
- **atz** Reset modem to default settings as determined by switches.
- **ATE0q1C1** No echo, no result codes, carrier enabled.
- **ats0=1** Auto answer after first ring.

chat If **aa**=TRUE, then **string** (in **chat** = **string**) will be output.

enabled The **getty** program uses this attribute to determine if special printing of the login prompt is needed. If the port should permit a logger, the value may be TRUE, SHARE, or DELAY. (To disable a port, use the **devices** command as described in the *AIX Operating System Commands Reference*.) Normally the value of TRUE is used if the port is to be enabled; however, if the port is to be shared (bi-directional use), then the value should be SHARE or DELAY. The value SHARE is used to make the port bi-directional with the device-locking scheme used by **uucp**, **cu**, **ate** and **connect**. DELAY operates like SHARE except that one or more characters must be read from the port by the **getty** process before the login herald will be printed. DELAY is useful with direct connections and intelligent modems. The **penable**, **pshare**, **pdelay**, **pdisable**, and **phold** commands will override the value specified by directly modifying **/etc/inittab** and sending a signal to the **init** process.

eof An octal integer specifying the character code that causes an end of file to be generated from the terminal. The system default is 004 (or 0x04), the ASCII EOT character, which is generated by **Ctrl-D**.

eol An optional and seldom-used alternate line termination character to use in addition to the ASCII new-line (line-feed) character.

erase An octal integer specifying the character code that deletes the previously received character. The system default for the erase character is 010 (or 0x08), **Ctrl-h**, which is generated by the **Backspace** key on many terminals.

herald An arbitrary string, enclosed in double quotes, printed by the **getty** process to prompt for login. The C language \(backslash) escapes **\r**, **\n**, **\t**, **\b**, and **\f** are recognized as carriage return, new-line, tab, backspace, and formfeed, respectively.

imap This attribute is used by **getty** to set the terminal input map.

If **imap** is not specified, **getty** resets the map to the system default.

intr An octal integer specifying the character code that interrupts the running process. The system default is 0177 (or 0x7f), which is usually generated by a key labeled **Del** or **Rubout**.

kill An octal integer specifying the character code that deletes the input line. The system default for the kill character is 025 (or 0x15), **Ctrl-u**, which is the ASCII NAK character.

lang This parameter defines the default value for the **LANG** environment variable of the programs started on the specified port.

lock This attribute is used to request port locking. If the value is TRUE, **init** creates a file in **/etc/locks** when the port is enabled and deletes the lock file when the port is disabled. Similarly, **penable** does not enable a port whose **lock** attribute is TRUE when the corresponding lock file exists. Programs using the port for some other purpose (such as a link between processors) should check for an outstanding lock (and create a lock file, if necessary) before opening the port.

log This parameter causes logins to be recorded for a port on the console or in file **/usr/adm/sulog**. If **log=true**, all logins are reported, and if **log=root**, logins by **root** (superuser) are recorded. See **super** parameter on 2.3.46.2 for related information.

logger A character string giving the names the program is to use at login. The default is **/bin/login**.

logmodes Console modes in effect while prompting for and reading in the user name. Modes are specified as a series of terminal options separated by a + (plus). Terminal options are as listed in the **stty** command. All listed modes not preceded with - (dash) are recognized. For example, the default **logmodes** parameter is specified as:

```
logmodes = cread+cs8+hupcl+echoe+echok
```

Because a speed value is not recognized in **logmodes** under any circumstances, the baud rate must be set with the **speed** parameter (see below).

min See the discussion of ICANON under "termio" in topic 2.5.28.

omap This attribute is used by **getty** to set the terminal output map. If **omap** is not specified, **getty** resets the map to the system default.

owner Normally, when a port is logged in, the **login** program sets the logged-in user to be the owner of that port. Specifying an owner (either a UID or user name), the system manager forces the **getty** process to set ownership even before opening the port.

parity The values ODD, EVEN, and NONE cause the generation of odd, even, and no parity, respectively, while **inpck**, **ignpar**, and

parmrk cause the checking input for parity errors, ignoring input characters with parity errors, and "marking" input parity errors as specified under "termio" in topic 2.5.28. These values can be combined, as in **parity=odd+inpck**.

program If a value is specified, it is taken as the name of a program to run immediately after setting the logmodes. This feature is useful for establishing special purpose server ports that respond to a connection with a special protocol handler. If the special assignment **program=HOLD** is specified, no program runs on the port, but the logmodes, ownership, and protection are set and the port is held open. This is useful to keep the desired modes associated with a port that is occasionally seized for some special purpose.

protection Normally the protection on terminal is set to **rw--w--w-** (octal 622 or 0x192). The protection parameter overrides this default. The value can be set to an octal mask or a string such as **rw-rw-rw-** (octal 666 or 0x1b6).

quit An octal integer specifying the character code that causes the running process to abort. The system default is 026 (or 0x16), which is generated by pressing **Ctrl-V**.

runmodes Console modes in effect after the user name is read. The mode in which the port is left, specified similar to **logmodes**.

speed A decimal integer from the set {50, 75, 110, 134.5, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600, 19200} depending on the hardware capability.

super This parameter is passed on the logger in its environment. If **super=false**, then **login** does not allow **root** (the superuser) to log in on the port. This is useful for security on off-site terminal connections such as telephone links. (See **log** parameter, on page 2.3.46.2.)

term This parameter is passed to the logger and shell in their environment (see "environment" in topic 2.4.6) in the variable TERM. Some application software uses this information to determine the type of terminal the user is using.

time See the discussion of ICANON under "termio" in topic 2.5.28.

timeout A decimal integer. If a user name is not specified before the given number of seconds, the **getty** process advances to the next port setting, or exits if all settings were exhausted.

Multiple values, separated by commas, can be specified as in the **speed=300,1200** line for dial-in terminals. This causes the port to be set up according to the first set of values for each attribute. If a framing error occurs, as a result of a user-generated BREAK on the line or a speed mismatch between the terminal and the set speed, the **getty** process advances to the next value on the list.

If multiple specifications occur for more than one parameter, all are advanced at the same time. Thus, a specification such as:

```
speed=300,1200
parity=none,odd+inpck
```

first tries the line at 300 baud with no parity. If a framing error occurs, it tries 1200 baud generation and checks for odd parity.

2.3.46.3 Other Port Parameters

The **ports** file has all the port-specific information, not just information about loggers. The other parameters in the file are:

loc The location of the terminal connected to the port. This parameter is presently unused by any AIX software. Because programs that access this file ignore keywords they do not use, helpful information can be added to keep all port-specific information together in one area.

printer The hard copy device used for output from optional word processing packages.

Example

The following example of a **ports** file illustrates some of its features:

```
  default:
       enabled = false
       speed = 9600
       herald = "\r\n\r\n ps2aix PS/2 login: "
       logmodes = echoe+hupcl+parenb+cs7
       runmodes = parenb+cs8+hupcl+cread+clocal+brkint+istrip
                  +icrnl+ixon+isig+icanon+echo+echoe+echok
                  +opost+onclr+tab3
       parity = true

  /dev/console:
       term = ibm 8513
       enabled = true
       herald = "\r\n\r\n ps2aix PS/2 Console login:"
```

Files

/etc/ports
/etc/locks

Related Information

In this book: "attributes" in topic 2.3.5, "connect.con" in topic 2.3.9, "environment" in topic 2.4.6, and "termio" in topic 2.5.28.

The **su, pstart, pdisable, getty, login, init**, and **stty** commands in *AIX Operating System Commands Reference*.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

2.3.47 predefined

Purpose
Provides information for predefined devices.

Description
The **predefined** file contains information about hardware adapters and devices that is used by the **devices** command. Some of these devices may not be present in a particular configuration, but all of them are supported by the system. The **predefined** file contains information needed when adding one of these devices so that you do not have to supply the information yourself. The size of this file increases with new entries as additional licensed programs are installed in the system.

The **devices** command uses the information in this file to set up stanzas in the **system** and **qconfig** files when devices are added to the system. Note that information in this file has no effect on the system until it is moved to a stanza in the **system** or **qconfig** file.

The **predefined** file is similar in structure and content to the **system** file, and its stanzas can contain any of the keywords that are allowed in the **system** file.

The use of extended characters in the **predefined** file is not supported.

The **predefined** file contains several special stanzas:

defqueue Used by the **devices** command to create the queue stanza in the **qconfig** file when a printer or plotter is added.

defdevice Used by the **devices** command to create the device stanza in the **qconfig** file when a printer or plotter is added.

default Contains keywords and their values that are common to all device stanzas.

ports Contains a list of ports supported by devices. A port is an adapter or a communications port, for example, an RS232 port, a parallel port, the PS/2 Tape Adapter, or the 3270 Connection Adapter. A port is described by the following lines:

```
        xyzn = minor
*         port description
```

where:

xyz is a unique port prefix, consisting of letters only

n is any positive decimal number

minor is the port's minor number

port description
is less than 20 characters in length. Port description must be the same length as the line that follows the **pflag** parameter in a device's **predefined** stanza that references **xyzn**.

n and **minor** do not have to be equal. If there are multiple ports with the same port prefix, the value of **n** must increment

from one port to the next.
For example, AIX PS/2 supports three parallel ports. Their port descriptions are as follows:

```
  p1 = 0
* Parallel port 1
  p2 = 1
* Parallel port 2
  p3 = 2
* Parallel port 3
```

Note that the description of all ports of prefix **xyz** must be the same length.

adapts Assigns ports to adapters. Each line in the **adapts** stanza has the following format:

```
ID = portrange
```

where

ID is the adapter ID contained in the adapter's POS registers.

portrange
is the range of valid ports.

For example, the Dual Asyn Adapter (DAA) has an adapter ID equal to **eeff**. The two DAA ports may be assigned to any one of eight serial ports. Given that the port prefix for serial ports is **s**, the port assignment for the DAA is:

```
eeff = s1-8
```

where valid serial port names are:

```
s1, s2, s3, s4, s5, s6, s7, and s8.
```

An adapter ID can be assigned only one port range.

Example
The following shows sample entries of the **predefined** file.

```
  defqueue:
        argname = none
        device = none

  defdevice:
        file = /dev/none
        backend = /usr/lpd/piobe

  default:
        modes = rw-rw-rw-
        owner = root
  *
  ports:
        s1 = 0
  * serial port 1
        s2 = 1
  * serial port 2
```

```
        s3 = 2
* serial port 3
        s4 = 3
* serial port 4
        s5 = 4


* serial port 5
        s6 = 5
* serial port 6
        s7 = 6
* serial port 7
        s8 = 7
* serial port 8
        p1 = 0
* parallel port 1
*
adapts:
0    = s1-8      *serial port on planar
1    = p1-3      *parallel port on planar
eeff = s1-8      *serial ports on Dual Async Adapter
ddff = s1-8      *serial ports on internal modem

4202s:
* IBM 4202 Proprinter XL on a serial port
        name = 4202s
        driver = sa
        minor = c
        kaf_file = /etc/ddi/sprinter.kaf
        kaf_use = kserial
        file = /etc/ddi/sprinter
        par = false
        use = d4202s
        noddi = false
        dtype = printer
* Printer
        specproc = cfgaqcfg
        noduplicate = false
        dname = lp
        noshow = false
        pflag = true
* Serial port 1
        port = s1-8
        slot = 0
```

File

/etc/predefined

Related Information

In this book: "attributes" in topic 2.3.5 and "system" in topic 2.3.56.

2.3.48 profile

Purpose
Sets the **sh** user environment at login time.

Description
The **profile** file contains commands to be executed at login and variable assignments to be set and exported into the environment. The **/etc/profile** file contains commands executed by all users at login.

After the **login** program adds the LOGNAME (login name) and HOME (login directory) parameters to the environment, the **/bin/sh** program is executed (if so indicated in the **/etc/passwd** file). The **sh** command executes the commands in **/etc/profile** and then, if present, the commands in **$HOME/.profile** are executed. The **.profile** file is the individual user profile that overrides the variables set in the **profile** file and is used to tailor the user environment variables set in **/etc/profile**. The **.profile** file is often used to set exported environment variables and terminal modes. The person who customizes the system can use **adduser** to set default **.profile** files in each user home directory. Users can tailor their environment as desired by modifying their **.profile** file.

Example
The following example is typical of a **/etc/profile** file:

```
# Set file creation mask
unmask 022
# Tell me when new mail arrives
MAIL=/usr/mail/$LOGNAME
# Add my /bin directory to the shell search sequence
$HOME/bin:$PATH
# Set terminal type
TERM=hft
# Make some environment variables global
export MAIL PATH TERM
```

Files

$HOME/.profile
/etc/profile

Related Information
In this book: "passwd" in topic 2.3.44, "environment" in topic 2.4.6, and "TERM" in topic 2.4.26.

The **env, login, mail, sh**, **stty**, and **su** commands in *AIX Operating System Commands Reference*.

2.3.49 qconfig

Purpose
Configures a printer queueing system.

Description
The **/etc/qconfig** file describes the queues and devices available for use by the **print** command, which places requests on a queue, and the **qdaemon** command, which removes requests from the queue and processes them. The **/etc/qconfig** file is an attribute file.

Some stanzas in this file describe queues, and other stanzas describe devices. Every queue stanza requires that one or more device stanzas immediately follow it in the file. The first queue stanza describes the default queue. The **print** command uses this queue when it receives no queue parameter.

The name of a queue stanza must be one to three characters long. The following table shows some of the field names along with some of the possible values that appear in this file:

acctfile Identifies the file used to save print accounting information. FALSE, the default, indicates suppress accounting. If the named file does not exist, no accounting is done.

argname Identifies the queue name identifier that is used in the **print** command to specify the queue.

device Identifies the symbolic name that refers to the device stanza.

discipline Defines the queue serving algorithm. The default, **fcfs**, means first come first served, while **sjn** means shortest job next.

node Identifies, by name, the TCF (Transparent Computing Facility) cluster site which serves the queue. The special symbol **local** is used to describe a generic queue which runs on all the sites in the cluster. The **/etc/qconfig** file looks for the node name in the TCF cluster **sitenam** field of the **/etc/site** file.

friend Indicates whether the backend updates the status file and responds to terminate signals. TRUE is the default. FALSE indicates it does not.

up Defines the state of the queue. TRUE, the default, indicates that it is running. FALSE indicates that it is not running.

If a field is omitted, its default value is assumed. The default values for a queue stanza are:

```
friend     = TRUE
discipline = fcfs
up         = TRUE
acctfile   = FALSE
```

Also, the default **argname** value is the name of the stanza preceded by a **-** (minus). The **device** and **node** fields cannot be omitted.

The name of a device stanza is arbitrary. The fields that can appear in a stanza are:

access Specifies the type of access the backend has to the file specified by the **file** field. The value of **access** is **write** if the backend has write access to the file, or **both** if it has both read and write access. This field is ignored if the **file** field has the value FALSE.

align Specifies whether the backend sends a form-feed control before starting the job if the printer was idle. The default is FALSE.

backend Specifies the full path name of the backend, optionally followed by flags and parameters to be passed to it.

feed Specifies the number of separator pages to print when the device becomes idle, or the value **never**, which indicates that the backend is not to print separator pages.

file Identifies the special file where the output of backend is to be redirected. FALSE, the default, indicates no redirection. In this case, the backend opens the output file.

header Specifies whether a header page prints before each job or group of jobs. The default, **never**, indicates no header page at all, while **always** specifies a header page before each job, and **group** specifies a header before each group of jobs for the same user.

trailer Specifies whether a trailer page prints after each job or group of jobs. **never**, the default, means no trailer page at all. **always** means a trailer page after each job. **group** means a trailer page after each group of jobs for the same user.

The **qdaemon** places the information contained in the **feed, header, trailer**, and **align** fields into a status file that is sent to the backend. Backends that do not update the status file do not use the information it contains.

If a field is omitted, its default value is assumed. The **backend** field cannot be omitted. The default values in a device stanza are:

```
file    = FALSE
access  = write
feed    = never
header  = never
trailer = never
align   = FALSE
```

The **print** command automatically converts the ASCII **qconfig** file to binary when the binary version is missing or older than the ASCII version. The binary version is found in **/etc/qconfig.bin**.

Unlike the format of the **ports** file, the **qconfig** file format does not allow default stanzas.

Examples

1. The batch queue supplied with the AIX system might contain these stanzas:

```
bsh:
     argname = bsh
     friend = FALSE
```

```
            discipline = fcfs
            device = bshdev
            node = local

      bshdev:
            backend = /bin/sh
```

To run a shell procedure called **myproc** using this batch queue, enter:

```
print bsh myproc
```

The queuing system runs the files one at a time, in the order submitted. The qdaemon process redirects standard input, standard output, and standard error to the **/dev/null** file.

2. To allow two batch jobs to run at once:

```
      bsh:
            argname    = save
            friend     = FALSE
            discipline = fcfs
            node       = local
            device     = bsh1,bsh2

      bsh1:
            backend    = /bin/sh

      bsh2:
            backend    = /bin/sh
```

Files

/etc/qconfig
/etc/qconfig.bin
/usr/lpd/digest

Related Information

In this book: "attributes" in topic 2.3.5.

The **print**, **lp**, and **qdaemon** commands in *AIX Operating System Commands Reference*.

2.3.50 rasconf

Purpose
Defines the reliability, availability, and serviceability (RAS) configuration file.

Description
The **rasconf** file defines attributes of the reliability, availability, and serviceability (RAS) system. Initially, RAS logging is inactive and must be activated before any RAS data can be collected.

This attribute file consists of stanzas that govern the actions of daemons associated with individual RAS devices. Each stanza name is the name of the associated RAS device.

The following attributes are valid:

file = file Specifies the file into which the daemon will write the RAS information.

size = blocks Specifies the maximum size, in 1024-byte blocks, to which the daemon will allow the file to grow.

Example
A typical **rasconf** file can contain the following:

```
/dev/klog:
  file = /usr/adm/messages
  size = 100

/dev/error:
  file = /usr/adm/ras/errfile
  size = 50

/dev/trace:
  file = /usr/adm/ras/trcfile
  size = 80
  buffer = 6
```

File
/etc/rasconf

Related Information
In this book: "attributes" in topic 2.3.5, "error" in topic 2.5.7, "osm" in topic 2.5.20, and "trace" in topic 2.5.29.

The **errdemon**, **syslogd** and **trace** commands in *AIX Operating System Commands Reference*.

2.3.51 RPC

Purpose
Contains RPC program information.

Description
The **rpc** file contains user readable names that can be used in place of RPC program numbers. Each line has the following information:

Name of server for the RPC progra

RPC program numbe

Aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Example

Here is an example of the **/etc/rpc** file:

```
#
# rpc  1.1 LCC)  /* Modified: 13:45:24  12/22/88 */
#
portmapper         100000     portmap sunrpc
rstatd             100001     rstat rup perfmeter
rusersd            100002     rusers
nfs                100003     nfsprog
ypserv             100004     ypprog
mountd             100005     mount showmount
ypbind             100007
walld              100008     rwall shutdown
yppasswdd          100009     yppasswd
etherstatd         100010     etherstat
rquotad            100011     rquotaprog quota rquota
sprayd             100012     spray
3270_mapper        100013
rje_mapper         100014
selection_svc      100015     selnsvc
database_svc       100016
rexd               100017     rex
alis               100018
sched              100019
llockmgr           100020
nlockmgr           100021
x25.inr            100022
statmon            100023
status             100024
```

Files

/etc/rpc

Related Information
The **rpcinfo** command in the *AIX Operating System Commands Reference*.

2.3.52 sccsfile

Purpose
Contains the Source Code Control System (SCCS) information.

Description
The SCCS file is an ASCII file consisting of the following six logical parts:

checksum The sum value of all characters, except the characters in the first line.

delta table Information about each delta including type, SCCS identification (SID) date and time of creation, and comments.

user names Login names and numerical group IDs, or both, of users who are allowed to add or remove deltas from the SCCS file.

flags Definitions of internal keywords.

comments Descriptive information about the file.

body The actual text lines intermixed with control lines.

There are lines throughout an SCCS file that begin with the ASCII SOH (start of heading) character (octal 001). This character is called the **control character** and is represented graphically as @ (at sign) in the following text. Any line described in the following text not shown beginning with the control character cannot begin with the control character.

The *DDDDD* entries represent a 5-digit string (a number from 00000 to 99999).

The following describes each logical part of an **SCCS** file.

Subtopics

2.3.52.1 Checksum

The checksum is the first line of an SCCS file. The value of the checksum is the sum of all characters, except those of the first line. The **@h** designates a **magic number** of 064001 octal (or 0x6801). The format of the line is:

@h*DDDDD*

2.3.52.2 Delta Table

The delta table consists of a variable number of entries such as:

```
@sDDDDD/DDDDD/DDDDD
@d type SCCS_ID yr/mo/da hh:mm:ss pgmr DDDDD DDDDD
@i DDDDD...
@x DDDDD...
@g DDDDD...
@m MR_number
.
.
.
@c comments...
.
.
.
@e
```

@s The first line, which contains the number of lines inserted or deleted or unchanged, respectively.

@d The second line, which contains:

The type of delta. **D** designates normal delta and **R** designates removed.

The *SCCS_ID* (SID) of the delta.

The date and time the delta was created.

The login name that corresponds to the real user ID at the time the delta was created.

The serial numbers of the delta and its predecessor.

@i Contains the serial numbers of the deltas included. This line is optional.

@x Contains the serial numbers of deltas excluded. This line is optional.

@g Contains the serial numbers of the deltas ignored. This line is optional.

@m Optional lines, each one containing one modification request (MR) number associated with the delta.

@c Comment lines associated with the delta.

@e Ends the delta table entry.

2.3.52.3 User Names

The list of login names and numerical group IDs, or both, of users who can add deltas to the file, separated by new-line characters. The bracketing lines **@u** and **@U** surround the lines containing the list. An empty list allows any user to make a delta.

2.3.52.4 Flags

Flags are keywords used internally in the system. For more information about their use, see the **admin** command in *AIX Operating System Commands Reference*. The format of each flag line is:

```
@f flag  optional text
```

The following flags are defined:

```
@ft  type of program
@fv  program name
@fi
@fb
@fm  module name
@ff  floor
@fc  ceiling
@fd  default-sid
@fn
@fj
@fl  lock-releases
@fq  user defined
```

The flags are used as follows:

b Allows the use of the **-b** option on the **get** command to cause a branch in the delta tree.

c Defines the highest release number that can be retrieved by a **get** command for editing. This release number must be less than or equal to 9999, and its default value is 9999. This release number is called the ceiling release number.

d Defines the default SID to be used when one is not specified with a **get** command.

f Defines the lowest release number that can be retrieved by a **get** command for editing. This release number must be between 0 and 9999, and its default value is 1. This release number is called the floor release number.

i Controls the error warning message **No ID keywords**. When this flag is not present, this message is only a warning. When this flag is present, the file is not used and the delta is not made.

j Causes the **get** command to allow concurrent edits of the same base SID.

l Defines a list of releases that cannot be edited with **get** using the **-e** flag.

m Defines the first choice for the replacement text of the **%M%** identification keyword.

n Causes the **delta** command to insert a delta that applies no changes for those skipped releases when a delta for a new release is made. For example, delta 5.1 is made after delta 2.1, skipping releases 3 and 4. When this flag is omitted, it causes skipped releases to be completely empty.

q Defines the replacement for the **%Q%** identification keyword.

t Defines the replacement for the **%Y%** identification keyword.

v Controls prompting for MR numbers in addition to comments. If optional text is present, it defines an MR number validity checking program.

2.3.52.5 Comments

Typically, the comments section contains a description of the purpose of the file. Bracketing lines **@t** and **@T** surrounding text designate the comments section.

2.3.52.6 Body

The body section consists of control and text lines. Control lines begin with the control character, text lines do not. There are three kinds of control lines: **insert, delete,** and **end,** represented by:

@I *DDDDD*
@D *DDDDD*
@E *DDDDD*

respectively. The digit string is the serial number corresponding to the delta for the control line.

Related Information
The **admin, delta, get**, and **prs** commands in *AIX Operating System Commands Reference*.

2.3.53 sendmail.cf

Purpose
Contains **sendmail** configuration file data.

Description
The configuration file contains the configuration information for the **sendmail** program. The configuration information includes such items as the host name and domain and the **sendmail** rule sets. If you have TCF installed, the cluster name is also defined in this file.

The configuration file has three major purposes:

To initialize the environment for **sendmail** by setting the options

To rewrite addresses in messages by first mapping the addresses fro any format into a canonical form and then mapping the canonical form into the appropriate syntax for the receiving mailer.

To translate the address into the set of instructions needed t deliver the message.

The configuration file entries consist of lines, each of which begins with a single character command. Entries can continue onto multiple lines by placing blanks at the beginning of each subsequent line. Comments are included on lines beginning with the # (sharp sign). The commands and operands are:

CX word1 word2 ...
Defines the class, specified by **X**, of words to match on the left hand side of rewriting rules. Class specifiers may be any of the uppercase letters from the ASCII character set. Lowercase letters and special characters are reserved for system use.

DX value
Defines the macro specified by **X** and its associated **value**. A macro is named using a single character. The character may be any character from the ASCII character set, but user-defined macros can only use the uppercase letters. Lowercase letters and special characters are reserved for system use. Macros can be interpolated in most places using the escape sequence $**x**. See "Special Macros" in topic 2.3.53.1 for additional information.

FX filename [**format**]
Reads the elements of the class specified by **X** from **filename** using an optional **scanf** format specifier.

H[?mflags?]hdrname: htemplate
Defines the header format the **sendmail** program inserts into a message. Continuation lines are a part of the definition and write into the outgoing message. The **htemplate** is macro expanded before insertion into the message. If the **mflags** are specified, at least one of the specified flags must be stated in the mailer definition for this header to be automatically output. If one of these headers is in the input, the header writes into the output message regardless of these flags.

Mname, [**field=value]***
Defines a mailer where **name** is the name of the mailer (used internally only) and **field=value** defines attributes of the mailer. Allowable **fields** and **values** are:

P=value Defines the path name of the mailer, where **value** is the path name. SMTP internal mail uses a value of [IPC].

F=value Defines the special flags for this mailer, where **value** can be a string composed of the following:

C Saves the sender domain. For this function only, the sender domain is defined to be everything from the first @ (at sign) to the end of the sender address. This string is appended to header addresses which contain no @ whenever mail is sent to any mailer. This also applies to calculation of the **$g** macro and everything dependent on it. This flag offsets the SMTP **mail** and **rcpt** commands.

D Requires a **Date** header line.

e This mailer is expensive to connect to. Avoid connecting normally. Any necessary connection will occur during a queue run. See the **c** option on 2.3.53.

E Escape **From** lines to >**From** in message bodies.

f The mailer wants a **-f** (from) flag only if this is a network forward operation (for example, the mailer gives an error if the executing user does not have special permissions).

F Requires a **From** header line.

h Preserves uppercase in host names for this mailer.

I Uses SMTP to communicate with another **sendmail** and can use special protocol features.

l This mailer is local; final delivery is performed.

L Limits the line length of a text line to less than 1000 characters. Any leading dot duplicated due to the **X** flag is not included in the count. Only allows 7-bit data to pass either way through the mailer.

m This mailer can send to multiple users on the same host in one transaction. When a **$u** macro occurs in the **A** part of the mailer definition, that field will be repeated as necessary for all qualifying users.

M Requires a **Message_ID** header line.

n The AIX-style **From** line on the front of the message is not inserted.

N International Character Support. Only has meaning when used with the **L** flag. Allows 8-bit data to pass.

p Uses the return-path in the SMTP **MAIL FROM:** command

rather than just the return address.

P Requires a **Return_Path** header line.

r Same as option **f** except a **-r** flag is generated.

s Strips quote characters off of the address before calling the mailer.

S User ID is not reset before calling the mailer.

u Preserves uppercase in users names for this mailer.

U Requires **From** lines with UUCP-style remote from **host** on the end.

x Requires a **Full_Name** header line.

X This mailer uses the hidden-dot algorithm. (Any line beginning with a dot has an extra dot prepended. This ensures that the lines in the message containing a leading dot will not terminate the message prematurely. See the **sendmail -i** flag or the **config** option.

S=value The rewriting rule set to be used for sender addresses, where **value** is the rewriting rule set number.

R=value The rewriting rule set to be used for recipient addresses, where **value** is the rewriting rule set number.

A=arg Defines the argument string **arg** to exec the mailer with. Embedded spaces may be included. If embedded spaces are used, enclose the argument string with " (double quotes). For an SMTP mailer, **A=[IPC]**.

E=string Defines the **string** to use as an end-of-line indication. A string containing only new-line is the default.

M=length Defines the maximum message **length** to be sent to the mailer.

Ox[value]
Sets option **x** to **value**. If the option is a valued option, you must also specify **value**. Options may also be selected from the command using the **-o** flag of the **sendmail** command. The options and the possible values are described as follows:

Afile Uses the named **file** as the alias file.

Bc Sets the blank substitution character to the character specified in the parameter **c**. The **sendmail** program replaces unquoted spaces in addresses with this character. The supplied configuration file uses the . (period) for this character.

c If an outgoing mailer is marked as being expensive to use, this option causes **sendmail** to queue messages for that mailer program without sending them. The queue can be run later when costs are lowest or when the queue is large

enough to send the message efficiently.

dx Sets the delivery mode to **x**. Valid modes are:

- **b** Deliver in background (asynchronously). This is the default setting.
- **i** Deliver interactively (synchronously)
- **q** Queue the message only and deliver during queue run.

ex Sets error processing to mode **x**. Valid modes are:

- **e** Mails the error message to the user's mail box, but always exits with a 0 exit status (normal return).
- **m** Mails the error message to the user's mail box.
- **p** Displays the error message on the terminal (default).
- **q** Discards the error message and returns the exit status only.
- **w** Writes the error message to the terminal or mails it if the user is not logged in.

f Saves **From** lines at the front of messages. These lines are normally discarded. Causes all other headers to be regarded as part of the message body.

gN Sets the default group ID to use when calling mailers to the value specified by **N**.

hdir If TCF is installed, this defines the master spool directory.

Hfile Specifies the name of the SMTP help file.

i Does not interpret a . (period) on a line by itself as a message terminator. Removes the excess dot inserted by a remote mailer at the beginning of a line, if mail is received through SMTP. In addition, if receiving mail through SMTP, any dot at the front of a line followed by another dot is removed. This is the opposite of the action performed by the **X** mailer flag.

Ix Allows spaces as well as tabs to separate the LHS and RHS of rewrite rules. In both the LHS and RHS, **x** must be used in place of embedded spaces. The default for **x** is _ (underscore). All instances of **x** are changed to spaces after the LHS and RHS are separated by the **sendmail** program. This option allows rewrite rules to be modified using an editor that replaces tabs with spaces.

Ln Specifies the log level to be the value supplied in the **n** parameter. Each number in the following list includes the activities of all numbers of lesser value and adds the activity that it represents. Valid levels and the activities they represent are:

- **0** No logging
- **1** Major problems only
- **2** Message collections and failed deliveries

3 Successful deliveries

4 Messages being deferred

5 Placing messages in the queue

6 Unusual but benign incidents

9 Log internal queue ID to external message ID mappings

12 Several messages that are of interest when debugging

16 Verbose information regarding the queue.

m If the sender is in an alias expansion, also send to the sender.

Mx value Defines macro **x** to have **value**. This option is normally used only from the **sendmail** command line.

n Validates the RHS of aliases when performing the **newaliases** function.

Nnetname Sets the name of the host network to **netname**. The **sendmail** program compares the argument of an SMTP **HELO** command to **hostname.netname** (value of **hostname** comes from the kernel). If these values do not match, it adds the **hostname.netname** string to the **Received:** line in the message so that messages can be traced accurately.

o Indicates that this message may have old style headers. Without this option, the message has new style headers (commas instead of spaces between addresses). If this option is set, an adaptive algorithm correctly determines the header format in most cases.

Paddress Identifies the person who is to receive a copy of all returned mail.

qfactor Use **factor** to decide when to queue messages rather than send them. This value is divided by the difference between the current load average and the load average limit (see the **x** option below) to determine the maximum message priority that will be sent. The default value is 10000.

Qdir Sets the directory in which to queue messages. The directory will be created if it does not exist.

rtime Sets the timeout for reads from a mailer program to the value specified by **time**. If no timeout value is set, **sendmail** waits indefinitely for a mailer to respond.

Sfile Sets the mail statistics file to the **file**. Statistics are only collected if the file exists. This file must be created by the user.

s Enqueues before delivery, even when in immediate delivery mode.

Ttime Sets the timeout on messages in the queue to the specified **time**. After this interval, **sendmail** returns the message to the sender. The default is three days.

uN Sets the default user ID to use when calling mailers to the value specified by **N**.

v Run in verbose mode.

xlavg When the system load average exceeds **lavg**, queue messages instead of sending them. The default value is 8.

Xlavg When the system load average exceeds **lavg**, incoming SMTP connections are refused. The default value is 12.

yfactor **factor** is added to the priority of the message once for each recipient (lowering the priority of the message). Messages with many recipients are thus penalized. The default value is 1000.

Y The **sendmail** program delivers each message in the mail queue from a separate process. This option is not required and may increase overhead in the AIX environment.

zfactor **factor** is multiplied by the message class (determined by the **Precedence:** field in the header and the **P** lines in the configuration file) and subtracted from the message priority. Thus, messages with higher **Precedence:** values are favored.

Zfactor **factor** is added to the message priority every time a message is processed, decreasing its priority. In most situations, **factor** should be positive, since hosts that are down are usually down for a long time. The default value is 9000.

Pname=num
Defines values for the **Precedence:** field. When **name** is found in a **Precedence:** field, the message class is set to **num**. Higher numbers indicate higher precedence. Negative numbers indicate that error messages are not returned. The default **num** is 0. The precedence of mail is defined by a header of that name within the mail.

Rlhs rhs comments
Defines a rewriting rule. One or more tab characters separate the three fields of this command. If space characters are to be used, the configuration option **I** must be set. The fields may contain embedded spaces, unless the **I** option is set. If the **I** option is set, the embedded spaces must be represented by the character defined in **I**. After the fields are separated, the character representing the space is changed to an actual space.

Sx
Begins the definition of a rule set. If a rule set definition is begun more than once, the new definition overwrites the old one.

Tuser1 user2...
Defines system administrative (trusted) user IDs. These IDs have permission to override the sender address using the **-f** flag. There may be more than one ID specified per line.

Subtopics
2.3.53.1 Special Macros

2.3.53.1 Special Macros

Macros are interpolated using the construct $**x**, where **x** is the name of the macro to be interpolated. Lowercase letters are reserved to have special semantics, used to pass information in or out of the **sendmail** program.

The following macros must be defined to transmit information into the **sendmail** program.

e The SMTP entry message. This message is sent by the SMTP handler in the **sendmail** domain when the host connects to it.

j The official domain name for the site. This must be the first word in the **$e** macro. The domain name is a sequence of domain element strings, ordered from the most specific to the most general, separated by periods. The use of nicknames or aliases is not allowed. The maximum domain name length is 64 characters. The **$j** macro should use this format.

l The format of the AIX **From** line. This macro is usually a constant.

n The name of the daemon (for error messages). This macro is usually a constant.

o The set of operators in addresses. This macro consists of a list of characters considered to be tokens and separates tokens during parsing. For example, if **r** exists in the **$o** macro, the input, **address**, parses into three tokens: **add, r,** and **ess**. There are many internal hard-coded delimiters added to this list by **sendmail**. It is recommended that this list not be changed.

q The default format of the sender address.

Sendmail defines some macros for interpolation into argument variables for mailers or for other contexts. These macros are:

a The origination date in Arpanet form. **$a** contains the time extracted from the **Date** line of the message (if there is one). If the incoming message has no **Date:** line, the **$a** macro contains the current time.

b The current date in Arpanet form. **$b** equals the current date and time (used for postmarks).

c The ***hop count***. The hop count is the number of times the message has been processed. The **-h** flag of the command line or the number of **Received:** headers in the message determine the hop count.

d The date in AIX (**ctime**) format.

f The sender (from) address. The **$f** macro is the sender address as seen from the current host.

g The sender address relative to the receiver. When mailing to a specific host, the **$g** macro contains the address of the sender relative to the receiver. For example, if the user, **newton**, at system, **appletree**, sends a message to **chopin@piano**, the **$f** macro equals **newton** and the **$g** macro equals **newton@appletree**.

h The receiving host.

i The queue ID of the host. The **$i** macro is useful for tracking messages if put into the message ID line.

p The process ID of **sendmail**. **$p** and **$t** are used to create unique strings for the **Message_ID** field.

s The host name of the sender.

t A numeric representation of the current time. The macros, **$p** and **$t**, are used to create unique strings for the **Message_ID** field.

u The receiving user

v The version number of the **sendmail** program. The **$v** macro can be found in **Received:** header messages and is useful for debugging.

w The hostname of the local site and, if present, the address.

x The full name of the sender. The name is determined by one of the following: the full name passed as a flag to **sendmail**, the value found in the **Full_Name** line of the header, the value found in the comment field of a **From** line, or if the message originates locally, the full name found in **/etc/passwd**.

y The terminal ID of the sender.

z The home directory of the receiver.

Files

/usr/lib/sendmail.cf The **sendmail** configuration file.

/usr/lib/sendmail.cfDB The compiled version of the **sendmail** configuration file.

Related Information

The **sendmail** command in *AIX Operating System Commands Reference*.

2.3.54 site

Purpose
Stores information about TCF cluster sites.

Description
The file **/etc/site** is used primarily by the Transparent Computing Facility and contains information about each machine in the TCF cluster. If TCF is not installed, the **site** file has one entry for the local machine. This entry contains the name of the machine as entered during the installation of the AIX Operating System (this name is returned by the **uname** command with the **-n** flag and in **nodename** by the **uname** system call.

The **site** file is an ASCII file with one entry per line; each entry contains position-dependent fields in the following format:

sitenum:sitenam:locnam:cputype:comment:fullnam:speed

If TCF is not installed, **sitenam** is obtained during installation of the Operating System; the other fields assume default values. The fields are:

sitenum Site number of the cluster site.

sitenam One-word name of the cluster site.

locnam Name of the cluster site's local file system.

cputype CPU type of the cluster site, for instance, i386, i370, or xa370.

comment Any information pertinent to the cluster site. For example, this field may contain CPU model numbers or configuration.

fullnam Full descriptive name of the cluster site.

speed Relative speed of the CPU of the cluster site. Its value is meaningful only in relation to the entries for other cluster sites.

The **site** file access routines (see "sfent, sfnum, sfname, sfctype, sfxcode, setsf, endsf" in topic 1.2.257) should be used to access this file.

File

/etc/site Site description file.

Related Information
In this book: "sfent, sfnum, sfname, sfctype, sfxcode, setsf, endsf" in topic 1.2.257.

The **site** command in *AIX Operating System Commands Reference*.

2.3.55 sitegroup

Purpose
Site permission file.

Description
The **sitegroup** file contains two fields, which are separated by colons. The first field contains a site access group which may be placed in the **site_exec_perm** field of entries in the file **/etc/passwd**. The second field is divided into subfields, each separated by semicolons. Each field consists of a number (an integer) which is the site number on which the group can execute or log in.

Examples
The following example allows users in **sitegroup** group A to run programs or log in on sites 1, 2, and 17.

```
groupA:1;2;17
```

File

/etc/sitegroup

Related Information
In this book: "passwd" in topic 2.3.44.

2.3.56 system

Purpose
Identifies the system devices.

Description
The **system** file contains entries for currently configured real devices and virtual devices.

The **system** file is an attribute file containing stanzas that generally describe special files including information about AIX drivers or system parameters. See "attributes" in topic 2.3.5 for a description of attribute files.

Subtopics
2.3.56.1 Special File Stanzas
2.3.56.2 System Parameter Stanzas

2.3.56.1 Special File Stanzas

Each special file named in the **system** file refers to a device driver entry in the **master** file. The driver entries specify the AIX device drivers to be configured. All drivers needed for specified special files are included, and those drivers marked as mandatory.

The name of each stanza is the simple name of the special file.

The use of extended characters in the **system** file is not supported.

address The device address for the virtual device (AIX/370 only).

aflag Not used.

arg The device's name and minor number for the corresponding disk or minidisk where **arg** activities should be performed.

ctibuf Number of input buffers that should be available for the device simultaneously.

ctobuf Number of output buffers that should be available for the device simultaneously.

dname Indicates the prefix name that is used to create the name of the device stanza in the **/etc/system** file and the special file in the **/dev** directory. The **devices** command uses this value when it creates a stanza name for a new special file.

driver Identifies the associated driver in the **master** file. This is mandatory in all device stanzas.

dtype Specifies the class of the device. Examples of this are **printer** and **disk**. The **devices** command displays this value when asking the user to choose a device class. It also uses this value to construct a list of device classes.

dump The device's name and minor number for the corresponding disk or minidisk where **dump** activities should be performed.

dumplow Location of the number of blocks into the dump partition where core dump should be started.

features Device-dependent parameter which is interpreted by the device driver on a device-by-device basis (refer to description of individual device).

fd The name of the fixed disk that the minidisk resides on, such as **hdisk0** and **hdisk1**. Used by **minidisks**.

file Identifies the file that contains the stanzas included by the **use** attribute. This is the **/etc/ddi** file associated with the device.

files Maximum number of entries in the system file table.

gmounts Number of different file systems that can be mounted in the same cluster simultaneously.

kaf_file Indicates the name of the keyword **attribute** file to be used by the customization helper programs for the device described

in the device stanza.

kaf_use Indicates the name of the stanza in the **kaf_file** that contains information about the attributes for the device.

locsite Specifies the cluster site number.

minor Has a value of the form *cn*, where *c* is either **b** to denote a block device, or **c** to denote a character device. *n* is the minor device number.

modes Sets the protection bits for the special file, specified in the form *rwxrwxrwx*. Hyphens replace modes that are turned off, for example, **rw-r--r--**.

mounts Not used.

name Is a required keyword that identifies the device type. For example, 4202, 4201 for printers.

nchann Number of AIX/370 channels.

nob Number of Blocks - used by **minidisks**

noddi Indicates whether any device-dependent information is associated with the device. The value TRUE indicates there is none. If **noddi=**TRUE, then the **change** subcommand of the **devices** command does not allow the user to change device characteristics. Although tape, Ethernet, and Token Ring have **ddi** files, **noddi** indicates that there is no information in them.

nodelete Indicates whether to delete the special file when this driver is removed. When this value is TRUE, no attempt is made to delete the special file.

nodl Indicates whether the device can be deleted from the system by the **devices** command. The value TRUE indicates the device cannot be deleted using this command.

noduplicate Indicates whether another device of this type can be added to the system. The value TRUE indicates another device cannot be added.

noipl Indicates whether this stanza is processed at initial program load (IPL) time. When this value is TRUE, this stanza is not processed at system initial program load (IPL) time.

noshow Indicates whether the **devices** command displays information from the stanza to the user. If **noshow=**FALSE, then the **showdev** subcommand of the **devices** command displays all device characteristics and the **showall** subcommand displays the device.

nospecial When this value is TRUE, no special file (**/dev** file) is to be created.

owner Specifies the name of the owner assigned to the **/dev** special file when it is created.

par A required keyword for printer stanzas. Indicates if the printer attaches to a parallel (TRUE) or serial (FALSE) port.

pflag Is a required keyword that indicates whether there is a port associated with the device. If the value is TRUE, then the **devices** command constructs the name of the **ddi** stanza when adding the device by concatenating the value of the **use** keyword and the port name.

If FALSE, then **devices** constructs the name of the **ddi** stanza by concatenating the value of the **use** keyword and a port number. A **maxminor** keyword must be defined if the value of **pflag** is FALSE.

The **showall** subcommand of the **devices** command displays the comment line that immediately follows the **aflag** definition as a description of the port. This comment line must have the same length as those that describe the relevant ports in the **ports** stanza.

pipe The device's name and minor number for the corresponding disk or minidisk where **pipe** activities should be performed.

port A required keyword for all stanzas with **pflag** set to TRUE. When **port** is used in an **/etc/predefined** stanza, it defines the range of valid ports for this device, like port ranges specified for adapters in the **adapts** stanza.

When a device, that is assigned a **port**, is added to **/etc/system**, **devices** removes the port range from the right side of the equal sign and replaces it with a port number in the port range.

root The device's name and minor number for the corresponding disk or minidisk where **root** activities should be performed.

slot Defines the slot number associated with a device. When defined in **/etc/predefined**, **slot** must be set to 0 if **pflag** is TRUE or --- if **pflag** is FALSE. If **pflag** is TRUE, **devices** will assign **slot** a valid slot number when adding the stanza to **/etc/system**.

specproc Indicates the name of the special processing routine that is to be invoked when customizing the system for the device. See "cfgadev" in topic 1.2.31 for information about the application program interface to this feature.

swap The device's name and minor number for the corresponding disk or minidisk where **swap** activities should be performed.

ttype Specifies the value to be used by **config** in the device-type field when building device configuration table for devices. When present, **ttype** overrides the contents of the drive value.

uinfo Specifies the hexadecimal bytes to pass to the **CFUDRV** type **ioctl** call to configure an AIX device driver. If a customization helper program is invoked, this attribute is not used.

units Number of devices; the number must be specified for each type of device available.

use Identifies a stanza to be logically included in the current stanza. If a **file** attribute is present, the file is searched to find the indicated stanza for device dependent information. This keyword is required if the **file** keyword is present.

vbuf Not used.

2.3.56.2 System Parameter Stanzas

All system parameters that are listed in **/etc/master** may have their default values overridden in the **sysparms** stanza. The format of system parameter keywords in **sysparms** is:

parm = **value**

Where:

parm Is a stanza name of a system parameter stanza in **/etc/master**

value Is the new value of the system parameter

Refer to **/etc/master** for a definition of all system parameters.

Other parameters can be given for special **customization helper** programs.

Example

The following is an excerpt of the **system** file entries:

```
  Smallest possible stanza:

  error:
          driver = err
          nospecial = true
          noshow = true

  Sample stanza with pflag set to false
  tokennet0:
  * Token Ring Device Driver
          name = token
          driver = token
          minor = c0
          kaf_file = /etc/ddi/token
          kaf_use = ktoken
          file = /etc/ddi/token
          noddi = true
          dtype = lan
  * Local Area Network
          pflag = false
  * Token Ring Device Driver
          modes = rw-rw-rw-
          noshow = false
          slot = ---
          nospecial = true
          specproc = /etc/lanspecial
          owner = root
          dname = tokennet0
          use = dtokennet0

  Sample stanza with pflag set to true (tty connected to a Dual Async
  Adapter port):
  tty0:
  * Asynchronous Terminal
          name = tty
          driver = sa
          minor = c0
          kaf_file = /etc/ddi/tty.kaf
          kaf_use = ktty
          file = /etc/ddi/tty
```

```
        noddi = true
        noduplicate = false
        dtype = ttydev
* Asynchronous Terminal
        specproc = cfgaport
        noshow = false
        pflag = true
* Serial port 1
        port = s1
        slot = 03
        noipl = false
        modes = rw-rw-rw-
        owner = root
        dname = tty0
        use = dttys1
```

File

/etc/system

Related Information

In this book: "attributes" in topic 2.3.5, "ddi" in topic 2.3.13, "master" in topic 2.3.32, and "predefined" in topic 2.3.47.

The **config**, **devices**, and **osconfig** commands in *AIX Operating System Commands Reference*.

2.3.57 System.Netid

Purpose
Describes RSCS connection information for AIX/370.

Description
The **/etc/System.Netid** file describes the connection between an AIX/370 guest and the RSCS mail and file transfer guest on the same VM host. RSCS is responsible for transmitting files or mail across a VM network or to the target guest machine on the same VM host.

Each AIX/370 system in a TCF cluster will maintain its own copy of this file. **/etc/System.Netid** is established as a symbolic link to **<LOCAL>/System.Netid**.

Lines of the file beginning with a "#" (pound sign) are treated as comments and are ignored. The rest of the file should consist of a single line with the six fields listed below; fields must be separated from one another by single spaces.

VM-nodeid AIX/370-guest AIX/370-Nodeid RSCS-Userid Class Hold_class

Where:

VM-nodeid Is the the node ID of the local VM system.

AIX/370-guest Is the user ID of the AIX/370 guest virtual machine. This should be the site name of the AIX/370 virtual machine.

AIX/370-Nodeid Is the node ID used to identify this AIX/370 system to the RSCS network. This should also be the site name of the AIX/370 virtual machine.

RSCS-Userid Is the user ID of the RSCS virtual machine used for network control. This is usually RSCS or NET. Consult your VM administrator for assistance.

Class Is the CP spool class used to mark files for this AIX/370 System. Class A is often chosen as the spool class for AIX/370.

Hold_class Is the CP spool class used to mark files put on temporary hold by the **rdrdaemon** command. Class B is often chosen as the hold spool class for AIX/370.

Example
The following is an example of a **System.Netid** file for the AIX/370 machine **AIX370** on VM with the node ID **ABCVM1**:

```
ABCVM1 AIX370 AIX370 RSCS A B
```

Files which cannot be delivered due to temporary conditions (for example, if the user's **$HOME/netfile** directory is unavailable) are put in a 'hold' state and delivery is retried approximately every 30 minutes. The values chosen for the **Class** and **Hold_class** fields must be different.

Related Information
The **rdrdaemon**, **uvcp**, and **vucp** commands in the *AIX Operating Systems Commands Reference*.

2.3.58 tar

Purpose
Describes the tape archive format.

Description
The **tar** command reads and writes tapes in tape archive format. A **tar** tape consists of several 512-byte logical blocks that can be grouped (on magnetic tape) into records, which are some constant multiple of 512-byte blocks long. Block in the following description means logical block.

The following is the format of a file header that precedes each disk file written on the tape:

```
struct {
     char name[100];
     char mode[8];
     char uid[8];
     char gid[8];
     char size[12];
     char mtime[12];
     char chksum[8];
     char typeflag;
     char linkname[100];
     char magic[6];
     char version[2];
     char uname[32];
     char gname[32];
     char devmajor[8];
     char devminor[8];
     char prefix[155];
};
```

All fields, except **typeflag**, are ASCII null-terminated strings. Numeric fields can contain leading blanks. The fields have the following meanings:

chksum Contains a byte-by-byte sum of the entire header block assuming that the **chksum** field is all blanks.

gid Contains the group identification of the file, in octal.

typeflag

Contains '0' or '\0' for a regular file.
Contains '1' if this file is a link to a previous file on the file on the tape.
Contains '2' for a symbolic link.
Contains '3' for a character special file.
Contains '4' for a block special file.
Contains '5' for a directory.
Contains '6' for a FIFO special file.

linkname Contains the name of a file if **linkflag** has a value of 1. The file named in this field is linked to the **name** file.

mode Contains the mode of the file, which includes the protection bits, setuid bits, setgid bits, and file type, in octal.

mtime Contains the modification time, in octal. This field gives the

major/minor device number and device site number for special files (the device site number is used by TCF).

name Contains the name of the file.

size Contains the size in bytes, in octal. This field is 0 for special files.

uid Contains the user identification of the file, in octal.

magic Contains the string TMAGIC ("ustar") indicating that this archive has been written in the POSIX-extended **tar** format.

version Contains the string TVERSION ("00").

uname Contains the ASCII representation of the owner of the file.

gname Contains the ASCII representation of the group of the file.

devmajor Contains the device major number for character and block special files.

devminor Contains the device minor number for character and block special files.

prefix Contains the path prefix of the archive file.

Unused bytes are null. Following the file header block are the data blocks of the file. The last block is null-padded if necessary. Two null blocks designate the end of the tape.

Directories and special files are treated in a slightly different way. A directory size is 0, meaning no data blocks follow, and its name ends with a / (slash). Also, if the directory is a hidden directory, an @ (at) is appended to the name prior to the / (slash). A special file is also written with 0 size. Its major/minor device number and device site number are in the **mtime** field.

File

/usr/include/tar.h

Related Information
The **tar** command in *AIX Operating System Commands Reference*.

2.3.59 terminfo

Purpose
Describes terminals by capability.

Description
A **terminfo** file is a data base that describes terminals, defining their capabilities and their methods of operation. It is used by various programs, including the Extended Curses Library (**libcur.a**) and the **vi** editor. The information defined includes initialization sequences, padding requirements, cursor positioning, and other command sequences that control specific terminals.

This section explains the **terminfo** source file format. Before a **terminfo** source file can be used, it must be compiled using the **tic** command, which is described in *AIX Operating System Commands Reference*. You can edit and modify these source files, such as **/usr/lib/terminfo/ibm.ti**, which describes IBM terminals, and **/usr/lib/terminfo/dec.ti**, which describes DEC terminals.

See "TERM" in topic 2.4.26 for a list of some of the terminals supported by predefined **terminfo** data base files and the corresponding values for the **TERM** environment variable.

If you have a terminal which is not supported, see if one of the many sample **terminfo** files supplied with your system will meet your need; if such a sample file exists, you can install it on your system (you may have to make some minor adjustments) using the "tic" program supplied with your system. Also, if you have **termcap** descriptions from other UNIX systems, you may translate them into **terminfo** descriptions, using an "ex" script called **/usr/lib/terminfo/cvt.ex** (supplied with your system, as well).

Each **terminfo** entry consists of a number of fields separated by commas, ignoring any white space between commas. The first field for each terminal gives the various names the terminal is known separated by **|** (vertical bar) characters. The first name given should be the most common abbreviation for the terminal, the last name given should be a long name fully identifying the terminal, and all others are understood as synonyms for the terminal name. All names except the last should be in lowercase and not contain blanks. The last name can contain uppercase characters for readability.

Terminal names (except the last) should be chosen using the following conventions. A root name should be chosen to represent the particular hardware class of the terminal. This name should not contain hyphens, except to avoid synonyms that conflict with other names. Possible modes for the hardware or user preferences are indicated by appending a **-** (hyphen) and an indicator of the mode to the root name. Thus, a terminal in 132 column mode would be **term-w**. The following suffixes should be used where possible:

Suffix	**Meaning**	**Example**
-am	With automatic margins (usually default)	**term**-am
-c	Color mode	**term**-c
-w	Wide mode (more than 80 columns)	**term**-w
-nam	Without automatic margins	**term**-nam
-**n**	Number of lines on the screen	**term**-60
-na	No arrow keys (leave them in local)	**term**-na
-**n**p	Number of pages of memory	**term**-4p

-rv Reverse video **term**-rv

Subtopics
2.3.59.1 Types of Capabilities
2.3.59.2 List of Capabilities
2.3.59.3 Preparing Descriptions
2.3.59.4 Basic Capabilities
2.3.59.5 Parameterized Strings
2.3.59.6 Cursor Motions
2.3.59.7 Area Clears
2.3.59.8 Insert/Delete Line
2.3.59.9 Insert/Delete Character
2.3.59.10 Highlighting, Underlining, and Visual Bells
2.3.59.11 Keypad
2.3.59.12 Tabs and Initialization
2.3.59.13 Miscellaneous Strings
2.3.59.14 Indicating Terminal Problems
2.3.59.15 Similar Terminals
2.3.59.16 Data Base File Names

2.3.59.1 Types of Capabilities

Capabilities in **terminfo** are of three types: boolean, numeric, and string. Boolean capabilities indicate that the terminal has some particular feature. Boolean capabilities are true if the corresponding name is in the terminal description. Numeric capabilities give the size of the terminal or the size of particular delays. String capabilities give a sequence that can be used to perform particular terminal operations.

Entries can continue onto multiple lines by placing white space at the beginning of each subsequent line. Comments are included on lines beginning with the # (sharp sign) character.

2.3.59.2 List of Capabilities

The following table shows **VARIABLE**, which is the name the programmer uses to access the **terminfo** capability. The **CAP NAME** (capability name) is the short name used in the text of the data base, and is used by a person updating the data base. The **I. CODE** is the 2-letter internal code used in the compiled data base, and always corresponds to a **termcap** capability name.

Capability names have no absolute length limit. An informal limit of five characters is adopted to keep them short and to allow the tabs in the source file **caps** to be aligned. Whenever possible, names are chosen to be the same as or similar to the ANSI X3.64 standard of 1979.

(P) Indicates that padding may be specified.

(G) Indicates that the string is passed through **tparm** with parameters as given (#**i**).

(*) Indicates that padding may be based on the number of lines affected.

(#**i**) Indicates the **i**(th) parameter.

VARIABLE	CAP NAME	I. CODE	DESCRIPTION
Booleans:			
auto_left_margin	bw	bw	Indicates **cub1** wraps from column 0 to last column.
auto_right_margin	am	am	Indicates terminal has automatic margins.
beehive_glitch	xsb	xs	Indicates a terminal with fl=escape and f2=Ctrl-C.
ceol_standout_glitch	xhp	xs	Indicates standout not erased by overwriting.
eat_newline_glitch	xenl	xn	Ignores new-line character after 80 columns.
erase_overstrike	eo	eo	Erases overstrikes with a blank.
generic_type	gn	gn	Indicates generic line type (such as, dialup, switch)
hard_copy	hc	hc	Indicates hardcopy terminal.
has_meta_key	km	km	Indicates terminal has a meta key (shift, sets parity bit).
has_status_line	hs	hs	Indicates terminal has extra "status line".
insert_null_glitch	in	in	Indicates insert mode distinguishes nulls.
memory_above	da	da	Retains information above display in memory.
memory_below	db	db	Retains information below display in memory.
move_insert_mode	mir	mi	Indicates safe to move while in insert mode.
move_standout_mode	msgr	ms	Indicates safe to move in standout modes.
over_strike	os	os	Indicates terminal overstrikes.

status_line_esc_ok	eslok	es	Indicates escape can be used on the status line.
teleray_glitch	xt	xt	Indicates destructive tabs and blanks inserted while entering standout mode.
tilde_glitch	hz	hz	Indicates terminal cannot print ~ characters.
transparent_underline	ul	ul	Overstrikes with underline character.
xon_xoff	xon	xo	Indicates terminal uses xon/xoff handshaking.

Numbers:

columns	cols	co	Specifies the number of columns in a line.
init_tabs	it	it	Provides tabs initially every # spaces.
lines	lines	li	Specifies the number of lines on screen or page
lines_of_memory	lm	lm	Specifies the number of lines of memory if > lines. A value of 0 indicates variable.
magic_cookie_glitch	xmc	sg	Indicates number of blank characters left by **smso** or **rmso**.
padding_baud_rate	pb	pb	Indicates lowest baud where carriage return and line return padding is needed.
virtual_terminal	vt	vt	Indicates virtual terminal number.
width_status_lines	wsl	ws	Specifies the number of columns in status line.

Strings:

appl_defined_str	apstr	za	Application defined terminal string.
back_tab	cbt	bt	Back tab. (P)
bell	bel	bl	Produces an audible signal (bell). (P)
box_chars_1	box1	bx	Box characters primary set.
box_chars_2	box2	by	Box characters alternate set.
box_attr_1	batt1	Bx	Attributes for box_chars_1.
box_attr_2	batt2	By	Attributes for box_chars_2.
carriage_return	cr	cr	Indicates carriage return. (P*)
change_scroll_region	csr	cs	Changes scroll region to lines #1 through #2. (PG)
clear_all_tabs	tbc	ct	Clears all tab stops. (P)
clear_screen	clear	cl	Clears screen and puts cursor in home position. (P*)
clr_eol	el	ce	Clears to end of line. (P)
clr_eos	ed	cd	Clears to end of the display. (P*)
color_bg_0	colb0	d0	Background color 0 black.
color_bg_1	colb1	d1	Background color 1 red.
color_bg_2	colb2	d2	Background color 2 green.
color_bg_3	colb3	d3	Background color 3 brown.
color_bg_4	colb4	d4	Background color 4 blue.

color_bg_5	colb5	d5	Background color 5 magenta.
color_bg_6	colb6	d6	Background color 6 cyan.
color_bg_7	colb7	d7	Background color 7 white.
color_fg_0	colf0	c0	Foreground color 0 white.
color_fg_1	colf1	c1	Foreground color 1 red.
color_fg_2	colf2	c2	Foreground color 2 green.
color_fg_3	colf3	c3	Foreground color 3 brown.
color_fg_4	colf4	c4	Foreground color 4 blue.
color_fg_5	colf5	c5	Foreground color 5 magenta.
color_fg_6	colf6	c6	Foreground color 6 cyan.
color_fg_7	colf7	c7	Foreground color 7 black.
column_address	hpa	ch	Sets cursor column. (PG)
command_character	cmdch	CC	Indicates terminal command prototype character can be set.
cursor_address	cup	cm	Indicates screen relative cursor motion row #1 col #2. (PG)
cursor_down	cud1	do	Moves cursor down one line.
cursor_home	home	ho	Moves cursor to home position (if no **cup**).
cursor_invisible	civis	vi	Makes cursor invisible.
cursor_left	cubl	le	Moves cursor left one space.
cursor_mem_address	mrcup	CM	Indicates memory relative cursor addressing.
cursor_normal	cnorm	ve	Makes cursor appear normal (undo **vs** or **vi**).
cursor_right	cuf1	nd	Indicates nondestructive space (cursor right).
cursor_to_ll	ll	ll	Moves cursor to first column of last line (if no **cup**).
cursor_up	cuu1	up	Moves cursor up one line (cursor up).
cursor_visible	cvvis	vs	Makes cursor very visible.
delete_character	dch1	dc	Deletes character. (P*)
delete_line	dl1	dl	Deletes line. (P*)
dis_status_line	dsl	ds	Disables status line.
down_half_line	hd	hd	Indicates subscript (forward 1/2 line feed).
enter_alt_charset_mode	smacs	as	Starts alternate character set. (P)
enter_blink_mode	blink	mb	Enables blinking.
enter_bold_mode	bold	md	Enables bold (extra bright) mode.
enter_ca_mode	smcup	ti	Begins programs that use **cup**.
enter_delete_mode	smdc	dm	Starts delete mode.
enter_dim_mode	dim	mh	Enables half-bright mode.
enter_insert_mode	smir	im	Starts insert mode.
enter_protected_mode	prot	mp	Enables protected mode.
enter_reverse_mode	rev	mr	Enables reverse video mode.
enter_secure_mode	invis	mk	Enables blank mode (characters invisible).
enter_standout_mode	smso	so	Begins standout mode.
enter_underline_mode	smul	us	Starts underscore mode.
erase_chars	ech	ec	Erases #1 characters. (PG)
exit_alt_charset_mode	rmacs	ae	Ends alternate character set. (P)
exit_attribute_mode	sgr0	me	Disables all attributes.
exit_ca_mode	rmcup	te	Ends programs that use **cup**.
exit_delete_mode	rmdc	ed	Ends delete mode.

exit_insert_mode	rmir	ei	Ends insert mode.
exit_standout_mode	rmso	se	Ends stand out mode.
exit_underline_mode	rmul	ue	Ends underscore mode.
flash_screen	flash	vb	Indicates visual bell (may not move cursor).
font_0	font0	f0	Select font 0.
font_1	font1	f1	Select font 1.
font_2	font2	f2	Select font 2.
font_3	font3	f3	Select font 3.
font_4	font4	f4	Select font 4.
font_5	font5	f5	Select font 5.
font_6	font6	f6	Select font 6.
font_7	font7	f7	Select font 7.
form_feed	ff	ff	Ejects page (hardcopy terminal). (P*)
from_status_line	fsl	fs	Returns from status line.
init_1string	is1	is	Initializes terminal.
init_2string	is2	is	Initializes terminal.
init_3string	is3	(none)	Initializes terminal.
init_file	if	if	Identifies file containing **is**.
insert_character	ich1	ic	Inserts character. (P)
insert_line	il1	al	Adds new blank line. (P*)
insert_padding	ip	ip	Inserts pad after character inserted. (P*)
key_backspace	kbs	kb	Sent by backspace key.
key_back_tab	kbtab	k0	Sent by backtab key.
key_catab	ktbc	ka	Sent by clear-all-tabs key.
key_clear	kclr	kC	Sent by clear-screen or erase key.
key_ctab	kctab	kt	Sent by clear-tab key.
key_command	kcmd	kc	Command request key.
key_command_pane	kcpn	kW	Command pane key.
key_dc	kdch1	kD	Sent by delete-character key.
key_dl	kdl1	kL	Sent by delete-line key.
key_do	kdo	ki	Do request key.
key_down	kcud1	kd	Sent by terminal down arrow key.
key_eic	krmir	kM	Sent by **rmir** or **smir** in insert mode.
key_end	kend	kw	End key.
key_eol	ke1	kE	Sent by clear-to-end-of-line key.
key_eos	ked	kS	Sent by clear-to-end-of-screen key.
key_f0	kf0	k0	Sent by function key F0.
key_f1	kf1	k1	Sent by function key F1.
key_f2	kf2	k2	Sent by function key F2.
key_f3	kf3	k3	Sent by function key F3.
key_f4	kf4	k4	Sent by function key F4.
key_f5	kf5	k5	Sent by function key F5.
key_f6	kf6	k6	Sent by function key F6.
key_f7	kf7	k7	Sent by function key F7.
key_f8	kf8	k8	Sent by function key F8.
key_f9	kf9	k9	Sent by function key F9.
key_f10	kf10	k;	Sent by function key F10.
key_f11	kf11	k<	Sent by function key F11.
key_f12	kf12	k>	Sent by function key F12.
key_help	khlp	kq	Help key.
key_home	khome	kh	Sent by home key.
key_ic	kich1	kI	Sent by insert character/enter

			insert mode key.
key_il	kil1	kA	Sent by insert line key.
key_left	kcub1	kl	Sent by terminal left arrow key.
key_ll	kll	kH	Sent by home-down key.
key_newline	knl	nl	New-line key.
key_next_pane	knpn	kv	Next-pane key.
key_npage	knp	kN	Sent by next-page key.
key_ppage	kpp	kP	Sent by previous-page key.
key_prev_cmd	kpcmd	kp	Sent by previous-command key.
key_quit	kquit	kQ	Quit key.
key_right	kcuf1	kr	Sent by terminal right arrow key.
key_scroll_left	kscl	kz	Scroll left.
key_scroll_right	kscr	kZ	Scroll right.
key_select	ksel	kU	Select key.
key_sf	kind	kF	Sent by scroll-forward/down key.
key_smap_in1	kmpf1	Kv	Input for special mapped key 1.
key_smap_out1	kmpt1	KV	Output for mapped key 1.
key_smap_in2	kmpf2	Kw	Input for special mapped key 2.
key_smap_out2	kmpt2	KW	Output for mapped key 2.
key_smap_in3	kmpf3	Kx	Input for special mapped key 3.
key_smap_out3	kmpt3	KX	Output for mapped key 3.
key_smap_in4	kmpf4	Ky	Input for special mapped key 4.
key_smap_out4	kmpt4	KY	Output for mapped key 4.
key_smap_in5	kmpf5	Kz	Input for special mapped key 5.
key_smap_out5	kmpt5	KZ	Output for mapped key 5.
key_sr	kri	kR	Sent by scroll-backward/up key.
key_stab	khts	kT	Sent by set-tab key.
key_tab	ktab	kn	Tab key.
key_up	kcuu1	ku	Sent by terminal up arrow key.
keypad_local	rmkx	ke	Ends keypad transmit mode.
keypad_xmit	smkx	ks	Puts terminal in keypad transmit mode.
lab_f0	lf0	l0	Labels function key F0 if not F0.
lab_f1	lf1	l1	Labels function key F1 if not F1.
lab_f2	lf2	l2	Labels function key F2 if not F2.
lab_f3	lf3	l3	Labels function key F3 if not F3.
lab_f4	lf4	l4	Labels function key F4 if not F4.
lab_f5	lf5	l5	Labels function key F5 if not F5.
lab_f6	lf6	l6	Labels function key F6 if not F6.
lab_f7	lf7	l7	Labels function key F7 if not F7.
lab_f8	lf8	l8	Labels function key F8 if not F8.
lab_f9	lf9	l9	Labels function key F9 if not

			F9.
lab_f10	lf10	la	Labels function key F10 if not F10.
meta_on	smm	mm	Enables "meta mode" (8th bit).
meta_off	rmm	mo	Disables "meta mode".
newline	nel	nw	Performs new-line function (behaves like CR followed by LF).
pad_char	pad	pc	Pads character (instead of NUL).
parm_dch	dch	DC	Deletes #1 characters. (PG*)
parm_delete_line	dl	DL	Deletes #1 lines. (PG*)
parm_down_cursor	cud	DO	Moves cursor down #1 lines. (PG*)
parm_ich	ich	IC	Inserts #1 blank characters. (PG*)
parm_index	indn	SF	Scrolls forward #1 lines. (PG)
parm_insert_line	il	AL	Adds #1 new blank lines. (PG*)
parm_left_cursor	cub	LE	Moves cursor left #1 spaces. (PG)
parm_right_cursor	cuf	RI	Moves cursor right #1 spaces. (PG*)
parm_rindex	rin	SR	Scrolls backward #1 lines. (PG)
parm_up_cursor	cuu	UP	Moves cursor up #1 lines. (PG*)
pkey_key	pfkey	pk	Programs function key #1 to type string #2.
pkey_local	pfloc	pl	Programs function key #1 to execute string #2.
pkey_xmit	pfx	px	Programs function key #1 to xmit string #2.
print_screen	mc0	ps	Prints contents of the screen.
prtr_off	mc4	pf	Disables the printer.
prtr_on	mc5	po	Enables the printer.
repeat_char	rep	rp	Repeats character #1 #2 times. (PG*)
reset_1string	rs1	r1	Resets terminal to known modes.
reset_2string	rs2	r2	Resets terminal to known modes.
reset_3string	rs3	r3	Resets terminal to known modes.
reset_file	rf	rf	Identifies the file containing reset string.
restore_cursor	rc	rc	Restores cursor to position of last **sc**.
row_address	vpa	cv	Positions cursor to an absolute vertical position (set row). (PG)
save_cursor	sc	sc	Saves cursor position. (P)
scroll_forward	ind	sf	Scrolls text up. (P)
scroll_reverse	ri	sr	Scrolls text down. (P)
set_attributes	sgr	sa	Defines the video attributes. (PG9)
set_tab	hts	st	Sets a tab in all rows, current column.
set_window	wind	wi	Indicates current window is lines #1-#2 cols #3-#4.
tab	ht	ta	Tabs to next 8-space hardware

			tab stop.
to_status_line	tsl	ts	Moves to status line, column #1.
underline_char	uc	uc	Underscores one character and moves beyond it.
up_half_line	hu	hu	Indicates superscript (reverse 1/2 line-feed).
init_prog	iprog	iP	Locates the program for init.
key_a1	ka1	K1	Specifies upper left of keypad.
key_a3	ka3	K3	Specifies upper right of keypad.
key_b2	kb2	K2	Specifies center of keypad.
key_c1	kc1	K4	Specifies lower left of keypad.
key_c3	kc3	K5	Specifies lower right of keypad.
prtr_non	mc5p	pO	Enables the printer for #1 bytes.

Terminal capabilities have names. For instance, the fact that a terminal has automatic margins (such as, an automatic new-line when the end of a line is reached) is indicated by the capability **am**. Hence the description of the terminal includes **am**. Numeric capabilities are followed by the # (sharp sign) character and then the value. Thus the cols#80 capability, which indicates the number of columns the terminal has, gives the value 80 for the terminal.

Finally, string-valued capabilities, such as **el** (clear to end of line sequence) are given by the 2-character code, an **=** (equal sign), and then a string ending at the following **,** (comma). A delay in milliseconds may appear anywhere in a string capability, enclosed between a **$<** and a **>** as in **el=\EK$<3>**, and padding characters are supplied by **tputs** to provide this delay. The delay can be either a number, such as 20, or a number followed by an ***** (asterisk), such as **3***. An asterisk indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. (In the case of insert character, the factor is still the number of **lines** affected. This is always 1, unless the terminal has **xenl** and the software uses it.) When an asterisk is specified, it is sometimes useful to give a delay of the form **a.b**, such as, 3.5, to specify a delay per unit to tenths of milliseconds. (Only one decimal place is allowed.)

A number of escape sequences are provided in the string-valued capabilities for easy encoding of characters there. Both **\E** and **\e** map to an **Escape** character, **^x** maps to a **Ctrl-x** for any appropriate x, and the sequences **\n, \l, \r, \t, \b, \f, \s** give a new-line, line-feed, return, tab, backspace, form-feed, and space. Other escapes include **\^** (backslash caret) for a **^** (caret), **\ ** (backslash backslash) for a **** (backslash), **\,** (backslash comma) for a **,** (comma), **\:** (backslash colon) for a **:** (colon), and **\0** (backslash) for the null character. (**\0** will produce **\200**, which does not terminate a string but behaves as a null character on most terminals.) Finally, characters can be given as 3 octal digits after a **** (backslash).

Sometimes, individual capabilities must be commented out. To do this, put a period before the capability name.

2.3.59.3 Preparing Descriptions

An effective way to prepare a terminal description is to imitate the description of a similar terminal in the **terminfo** file and add to the description gradually, using partial descriptions with **vi** to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of this file to describe it or bugs in **vi**. To test a new terminal description, set the environment variable TERMINFO to a path name of a directory containing the compiled description you are working on and programs will look there rather than in **/usr/lib/terminfo**. A test to get the correct padding (if not known) is to edit the **/etc/passwd** file at 9600 baud, delete about 16 lines from the middle of the screen, then hit the **u** key several times quickly. If the terminal fails to display the result properly, more padding is usually needed. A similar test can be used for insert character.

2.3.59.4 Basic Capabilities

The following describe basic terminal capabilities:

am Indicates that the cursor moves to the beginning of the next line when it reaches the right margin. This capability also indicates whether the cursor can move beyond the bottom right corner of the screen.

bel Produces an audible signal (such as a bell or a beep).

bw Indicates that a backspace from the left edge of the terminal moves the cursor to the last column of the previous row.

clear Clears the screen leaving the cursor in the home position.

cols Specifies the number of columns on each line for the terminal.

cr Moves the cursor to the left edge of the current row. This code is usually carriage return (**Ctrl-M**).

cub1 Moves the cursor one space to the left, such as backspace.

cuf1, cuu1, and **cud1**
Moves the cursor to the right, up, and down, respectively.

hc Specifies a printing terminal. The **os** capability should also be specified.

lines Specifies the number of lines on a cathode ray tube (CRT) terminal.

os Indicates that when a character is displayed or printed in a position already occupied by another character, the terminal overstrikes the existing character, rather than replacing it with the new character. **os** applies to storage scope, printing, and APL terminals.

The **terminfo** initialization subroutine, **setupterm**, calls **termdef** to determine the number of lines and columns on the display. If **termdef** cannot supply this information, then **setupterm** uses the **lines** and **cols** values in the data base.

A point to note here is that the local cursor motions encoded in **terminfo** are undefined at the left and top edges of a CRT terminal. Programs should never attempt to backspace around the left edge, unless **bw** is given, and never attempt to go up locally off the top. In order to scroll text up, a program should go to the bottom left corner of the screen and send the **ind** (index) string.

To scroll text down, a program goes to the top left corner of the screen and sends the **ri** (reverse index) string. The strings **ind** and **ri** are undefined when not on their respective corners of the screen.

The **am** capability tells whether the cursor sticks at the right edge of the screen when text is output, but this does not necessarily apply to a **cuf1** from the last column. Local motion is defined from the left edge only if **bw** is given. In this case, a **cub1** from the left edge moves to the right edge of the previous row. If **bw** is not given, the effect is undefined. This is useful for drawing a box around the edge of the screen, for

example. If the terminal has switch-selectable automatic margins, the **terminfo** file usually assumes that it is on by specifying **am**. If the terminal has a command that moves to the first column of the next line, that command can be given as **nel** (newline). It does not matter if the command clears the remainder of the current line, so if the terminal has no **cr** and **lf**, it may still be possible to craft a working **nel** out of one or both of them.

These capabilities suffice to describe printing terminals and simple CRT terminals. Thus, the Model 33 Teletype is described as:

```
  33 | tty33 | tty | Model 33 Teletype,
      bel=^G, cols#72, cr=^M,     cud1=^J, hc, ind=^J, os,
```

And another terminal is described as:

```
  xxxx | x | xxxxxxxx,
      am, bel=^G, clear=^Z, cols#80, cr=^M, cub1=^H, cud1=^J,
      ind=^J, lines#24,
```

2.3.59.5 Parameterized Strings

Cursor addressing and other strings requiring parameters in the terminal are described by a parameterized string capability, with escapes similar to **printf %x** in it. For example, to address the cursor, the **cup** capability is given using two parameters: the row and column to address to. (Rows and columns are numbered starting with 0 and refer to the physical screen visible to the user, not to any unseen memory.) If the terminal has memory relative cursor addressing, that can be indicated by **mrcup**.

The capabilities and their parameters, with descriptions, are:

cub1 Backspaces the cursor one space.

cup Addresses the cursor using two parameters: the row and column to address. Rows and columns are numbered starting with 0 and refer to the physical screen visible to the user, not to memory.

cuu1 Moves the cursor up one line on the screen.

hpa and **vpa**
Indicates the cursor has row or column absolute cursor addressing, horizontal position absolute (**hpa**) and vertical position absolute (**vpa**).

Sometimes the **hpa** and **vpa** capabilities are shorter than the more general two parameter sequence and can be used in preference to **cup**. If there are parameterized local motions (such as, move **n** spaces to the right) these can be given as **cud**, **cub**, **cuf**, and **cuu** with a single parameter indicating how many spaces to move. These are primarily useful if the terminal does not have **cup**.

indn and **rin**
Scrolls text. These are parameterized versions of the basic capabilities **ind** and **ri**. **n** is the number of lines.

mrcup Indicates the terminal has memory-relative cursor addressing.

The parameter mechanism uses a stack and special **%** codes to manipulate it. Typically a sequence pushes one of the parameters onto the stack and then prints it in some format. Often more complex operations are necessary.

The **%** encodings have the following meanings:

%%	Outputs a **%**. (percent sign).
%d	Print pop() as in **printf** (numeric string from stack).
%2d	Print pop() like **%2d** (minimum 2 digits output from stack).
%3d	Print pop() like **%3d** (minimum 3 digits output from stack).
%02d	Prints as in **printf** (2 digits output).
%03d	Prints as in **printf** (3 digits output).
%c	Print pop() gives **%c** (character output from stack).
%s	Print pop() gives **%s** (string output from stack).
%p[i]	Pushes the **i**(th) parameter onto stack.
%P[a-z]	Sets variable [**a-z**] to pop() (variable output from stack).
%g[a-z]	Gets variable [**a-z**] and pushes it onto the stack.
%'c'	Character constant **c**.
%{nn}	Integer constant **nn**.

%+ %- %* %/ %m
Arithmetic (**%m** is modulus): push(pop() **operation** pop())
%& %| %^ Bit operations: push(pop() **operation** pop())
%= %> %< Logical operations: push(pop() **operation** pop()).
%! %¬ Unary operations push(**operation** pop())
%i Add 1 to first two parameters (for ANSI terminals).

%? expr %t thenpart %e elsepart %;
If-then-else. The **%e elsepart** is optional. You can make an else-if construct as with Algol 68:

%? **c**[1] %t **b**[1] %e **c**[2] %t **b**[2] %e **c**[3] %t **b**[3] %e **b**[4] %;

In this example, **c[i]** denote conditions, and **b[i]** denote bodies.

Binary operations are in postfix form with the operands in the usual order. That is, to get **x - 5** one would use **%gx%{5}%-**.

Consider a terminal, which, to get to row 3 and column 12, needs to be sent **\E&a12c03Y** padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its **cup** capability is **cup=6\E&a%p2%2dc%p1%2dY**.

Some terminals need the current row and column sent preceded by a **^T** with the row and column simply encoded in binary; **cup=^T%p1%c%p2%c**. Terminals that use **%c** need to be able to backspace the cursor (**cub1**), and to move the cursor up one line on the screen (**cuu1**). This is necessary because it is not always safe to transmit **\n**, **^D**, and **\r**, as the system may change or discard them. (The library routines dealing with **terminfo** set terminal modes so that tabs are not expanded by the operating system; thus **\t** is safe to send.)

A final example is a terminal that uses row and column offset by a blank character, thus **cup=\E=%p1%' '%+%c%p2%' '%+%c**. After sending **\E=**, this pushes the first parameter, pushes the ASCII value for a space (32), adds them (pushing the sum on the stack in place of the two previous values) and outputs that value as a character. Then the same is done for the second parameter. More complex arithmetic is possible using the stack.

2.3.59.6 Cursor Motions

If the terminal has a fast way to home the cursor (to very upper left corner of screen) then this can be given as **home**. Similarly a fast way of getting to the lower left-hand corner can be given as **ll**; this may involve going up with **cuu1** from the home position, but a program should never do this itself (unless **ll** does) because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as addressing (0,0) to the top left corner of the screen, not of memory. (Thus, the **\EH** sequence on some terminals cannot be used for **home**.)

2.3.59.7 Area Clears

The following areas are used to clear large areas of the terminal:

ed Clears from the current position to the end of the display. This is defined only from the first column of a line. (Thus, it can be simulated by a request to delete a large number of lines, if a true **ed** is not available.)

el Clears from the current cursor position to the end of the line without moving the cursor.

2.3.59.8 Insert/Delete Line

The following describes the insert and delete line capabilities:

csr Indicates the terminal has a scrolling region that can be set. This capability takes two parameters: the top and bottom lines of the scrolling region.

da Indicates the terminal can retain display memory above what is visible.

db Indicates the display memory can be retained below what is visible.

dl1 Indicates the line the cursor is on can be deleted. This done only from the first position on the line to be deleted. Additionally, the **dl** capability takes a single parameter indicating the number of lines to be deleted.

il1 Creates a new blank line before the line where the cursor is currently located and scrolls the rest of the screen down. This is done only from the first position of a line. The cursor then appears on the newly blank line. Additionally, the **il** capability can take a single parameter indicating the number of lines to insert.

rc Restores the cursor. When used after the **csr** capability, it gives an effect similar to delete line.

sc Saves the cursor. When used after the **csr** capability, it gives an effect similar to insert line.

wind Indicates the terminal has the ability to define a window as part of memory. This a parameterized string with 4 parameters: the starting and ending lines in memory and the stating and ending columns in memory, in that order.

2.3.59.9 Insert/Delete Character

Generally, there are two kinds of programmable terminals with respect to insert/delete character operations which can be described using the **terminfo** file. The most common insert/delete character operations affect only the characters on the current line and shift characters to the right and off the line. Other terminals make a distinction between typed and untyped blanks on the screen, shifting data displayed to insert or delete at a position on the screen occupied by an untyped blank, which is either eliminated or expanded to two untyped blanks. Clearing the screen and then typing text separated by cursor motions differentiates between the terminal types. You can determine the kind of terminal you have by doing the following:

1. Type **abc def** using local cursor movements, not spaces, between the **abc** and the **def**.

2. Position the cursor before the **abc** and place the terminal in insert mode. If typing characters causes the characters on the line to the right of the cursor to shift and exit the right side of the display, the terminal does not distinguish between blanks and untyped positions. If the **abc** moves to positions to the immediate left of the **def** and the characters move to the right on the line, around the end, and to the next line, the terminal is the second type. This is described by the **in** capability, which signifies insert null.

While these are two logically separate attributes (one line vs. multiline insert mode, and special treatment of untyped spaces) there are no known terminals whose insert mode cannot be described with the single attribute.

The **terminfo** file can describe both terminals having an insert mode and terminals that send a simple sequence to open a blank position on the current line. The following are used to describe insert or delete character capabilities:

dch1 Deletes a single character. **dch** with one parameter, **n** deletes **n** characters.

ech Erases **n** characters (equivalent to typing **n** blanks without moving the cursor) with one parameter.

ich1 Precedes the character to be inserted. Most terminals with an insert mode do not use this. Terminals that send a sequence to open a screen position should give it. (If the terminal has both, insert mode is usually preferable to **ich1**. Do not give both unless the terminal actually requires both to be used in combination.)

ip Indicates post padding needed. This is given as a number of milliseconds. Any other sequence that may need to be sent after inserting a single character can be given in this capability.

mir Allows cursor motion while in insert mode. It is sometimes necessary to move the cursor while in insert mode to delete characters on the same line. Some terminals may not have this capability due to their handling of insert mode.

rmdc Exits delete mode.

rmir Ends insert mode.

smdc Enters delete mode.

smir Begins insert mode.

Note that if your terminal needs both to be placed in an insert mode and a special code to precede each inserted character, then both **smir/rmir** and **ich1** can be given, and both will be used. The **ich** capability, with one parameter, **n**, will repeat the effects of **ich1 n** times.

2.3.59.10 Highlighting, Underlining, and Visual Bells

If your terminal has one or more kinds of display attributes such as highlighting, underlining, and visual bells, these can be presented in a number of ways. Highlighting, such as standout mode, presents a good, high contrast, easy-on-the-eyes format to add emphasis to error messages, and other attention getters. Underlining is another method to focus attention to a particular portion of the terminal. Visual bells include methods such as flashing the screen. The following capabilities describe highlighting, underlining, and visual bells for a terminal:

blink Indicates terminal has blink highlighting mode.

bold Indicates terminal has extra bright highlighting mode.

civis Causes the cursor to be invisible.

cnorm Causes the cursor to display normal. This capability reverses the effects of the **civis** and **cvvis** capabilities.

cvvis Causes the cursor to be more visible than normal when it is not on the bottom line.

dim Indicates the terminal has half-bright highlighting modes.

eo Indicates blanks erase overstrikes.

flash Indicates the terminal has a way of flashing the screen (a bell replacement) for errors without moving the cursor.

invis Indicates the terminal has blanking or invisible text highlighting modes.

msgr Indicates it is safe to move the cursor while in standout mode. Otherwise, programs using standout mode should exit standout mode before moving the cursor or sending a new-line. Some terminals automatically leave standout mode when they move to a new line or the cursor is addressed.

prot Indicates the terminal has protected highlighting mode.

rev Indicates the terminal has reverse video mode.

rmso Exits standout mode.

rmul Ends underlining.

sgr Sets attributes. **sgr0** turns off all attributes. Otherwise, if the terminal allows a sequence to set arbitrary combinations of modes, **sgr** takes 9 parameters. Each parameter is either 0 or 1, as the corresponding attribute is on or off. The 9 parameters are in this order: standout, underline, reverse, blink, dim, bold, blank, protect, and alternate character set. (**sgr** can only support those modes for which separate attributes exist on a particular terminal.)

smcup and **rmcup**
Indicates the terminal needs to be in a special mode when running a program that uses any of the highlighting, underlining or visual bell capabilities. **smcup** enters this mode, while

rmcup exits this mode. This need arises, for example, from terminals with more than one page of memory. If the terminal has only memory relative cursor addressing, and not screen relative cursor addressing, a screen-sized window must be fixed into the terminal for cursor addressing to work properly. This is also used where **smcup** sets the command character to be used by the **terminfo** file.

smso Enters standout mode.

smul Begins underlining.

uc Underlines the current character and moves the cursor one space to the right.

ul Indicates the terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike.

xmc Indicates the number of blanks left if the capability to enter or exit standout mode leaves blank spaces on the screen.

2.3.59.11 Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local mode. If the keypad can be set to transmit or not transmit, give these codes as **smkx** and **rmkx**. Otherwise the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kcub1, kcuf1, kcuu1, kcud1,** and **khome**, respectively. If there are function keys such as **F0, F1,..., F10,** the codes they send can be given as **kf0, kf1,..., kf10**. If these keys have labels other than the default **F0** through **F10**, the labels can be given as **lf0, lf1,..., lf10**. The codes transmitted by certain other special keys can be given as:

kbs Indicates the **backspace** key.

kclr Indicates the **clear screen** or **erase** key.

kctab Indicates clear the tab stop in this column.

kdch1 Indicates the **delete character** key.

kdl1 Indicates the **delete line** key.

ked Indicates clear to end of screen.

kel Indicates clear to end of line.

khts Indicates set a tab stop in this column.

kich1 Indicates insert character or enter insert mode.

kil1 Indicates insert line.

kind Indicates scroll forward and/or down.

kll Indicates home down key (home is the lower left corner of the display, in this instance).

kmir Indicates exit insert mode.

knp Indicates next page.

kpp Indicates previous page.

ktbc Indicates the **clear all tabs** key.

ri Indicates scroll backward and/or up.

In addition, if the keypad has a 3-by-3 array of keys including the 4 arrow keys, the other 5 keys can be given as **ka1**, **ka3**, **kb2**, **kc1**, and **kc3**. These keys are useful when the effects of a 3-by-3 directional pad are needed.

2.3.59.12 Tabs and Initialization

If the terminal has hardware tabs, the command to advance to the next tab stop can be given as **ht** (usually **Ctrl-I**). A "backtab" command which moves left toward the previous tab stop can be given as **cbt**. By convention, if the terminal modes indicate that tabs are being expanded by the operating system rather than being sent to the terminal, programs should not use **ht** or **cbt** even if they are present, since the user may not have the tab stops properly set. If the terminal has hardware tabs that are initially set every **n** spaces when the terminal is powered up, the numeric parameter **it** is given, showing the number of spaces the tabs are set to. This is normally used by the **tset** command to determine whether to set the mode for hardware tab expansion, and whether to set the tab stops. If the terminal has tab stops that can be saved in nonvolatile memory, the **terminfo** description can assume that they are properly set.

Other capabilities include **is1**, **is2**, and **is3**, initialization strings for the terminal, **iprog**, the path name of a program to be run to initialize the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to set the terminal into modes consistent with the rest of the **terminfo** description. They are normally sent to the terminal, by the **tset** program, each time the user logs in. They are printed in the following order: **is1**, **is2**, setting tabs using **tbc** and **hts**; **if**; running the program **iprog**; and finally **is3**. Most initialization is done with **is2**. Special terminal modes can be set up without duplicating strings by putting the common sequences in **is2** and special cases in **is1** and **is3**. A pair of sequences that does a harder reset from a totally unknown state can be analogously given as **rs1**, **rs2**, **rf**, and **rs3**, analogous to **is2** and **if**. These strings are output by the **reset** program, which is used when the terminal starts behaving strangely, or not responding at all. Commands are normally placed in **rs2** and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set the terminal into 80-column mode would normally be part of **is2**, but it causes an annoying screen behavior and is not normally needed since the terminal is usually already in 80-column mode.

If there are commands to set and clear tab stops, they can be given as **tbc** (clear all tab stops) and **hts** (set a tab stop in the current column of every row). If a more complex sequence is needed to set the tabs than can be described by this, the sequence can be placed in **is2** or **if**.

Certain capabilities control padding in the terminal driver. These are primarily needed by hard copy terminals, and are used by the **tset** program to set terminal modes appropriately. Delays embedded in the capabilities **cr**, **ind**, **cub1**, **ff**, and **tab** cause the appropriate delay bits to be set in the terminal driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb**.

2.3.59.13 Miscellaneous Strings

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pad**. Only the first character of the **pad** string is used.

If the terminal has an extra "status line" that is not normally used by software, this fact can be indicated. If the status line is viewed as an extra line below the bottom line, into which one can cursor address normally, the capability **hs** should be given. Special strings to go to the beginning of the status line and to return from the status line can be given as **tsl** and **fsl**. (**fsl** must leave the cursor position in the same place it was before **tsl**. If necessary, the **sc** and **rc** strings can be included in **tsl** and **fsl** to get this effect.) The parameter **tsl** takes one parameter, which is the column number of the status line the cursor is to be moved to. If escape sequences and other special commands, such as tab, work while in the status line, the flag **eslok** can be given. A string that turns off the status line (or otherwise erases its contents) should be given as **dsl**. If the terminal has commands to save and restore the position of the cursor, give them as **sc** and **rc**. The status line is normally assumed to be the same width as the rest of the screen, such as, **cols**. If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded) the width, in columns, can be indicated with the numeric parameter **wsl**.

If the terminal can move up or down half a line, this can be indicated with **hu** (half-line up) and **hd** (half-line down). This is primarily useful for superscripts and subscripts on hardcopy terminals. If a hardcopy terminal can eject to the next page (form-feed), give this as **ff** (usually **Ctrl-L**).

If there is a command to repeat a given character a given number of times (to save time transmitting a large number of identical characters) this can be indicated with the parameterized string **rep**. The first parameter is the character to be repeated and the second is the number of times to repeat it. Thus, **tparm(repeat_char,'x',10)** is the same as **xxxxxxxxxx**.

If the terminal has a "meta key" which acts as a shift key, setting the eighth bit of any character transmitted, this fact can be indicated with **km**. Otherwise, software will assume that the eighth bit is parity and it will usually be cleared. If strings exist to turn this "meta mode" on and off, they can be given as **smm** and **rmm**.

If the terminal has more lines of memory than will fit on the screen at once, the number of lines of memory can be indicated with **lm**. A value of **lm#0** indicates that the number of lines is not fixed, but that there is still more memory than fits on the screen.

Media copy strings that control an auxiliary printer connected to the terminal can be given in the following ways: **mc0** prints the contents of the screen, **mc4** turns off the printer, and **mc5** turns on the printer. When the printer is on, all text sent to the terminal is sent to the printer. It is undefined whether the text is also displayed on the terminal screen when the printer is on. A variation **mc5p** takes one parameter, and leaves the printer on for as many characters as the value of the parameter, then turns the printer off. The parameter should not exceed 255. All text, including **mc4**, is transparently passed to the printer while an **mc5p** is in effect.

Strings to program function keys can be given as **pfkey**, **pfloc**, and **pfx**.

Each of these strings takes two parameters: the function key number to program (from 0 to 10) and the string to program it with. Function key numbers out of this range can program undefined keys in a terminal-dependent manner. The difference between the capabilities is that **pfkey** causes pressing the given key to be the same as the user typing the given string; **pfloc** causes the string to be executed by the terminal in local mode; and **pfx** causes the string to be transmitted to the computer.

2.3.59.14 Indicating Terminal Problems

Terminals that do not allow ~ (tilde) characters to be displayed should indicate **hz**.

Terminals that ignore a line-feed character immediately after an **am** wrap should indicate **xenl**.

If **el** is required to get rid of standout (instead of merely writing normal text on top of it), **xhp** should be given.

Terminals for which tabs turn all characters moved to blanks should indicate **xt** (destructive tabs). This capability is interpreted to mean that it is not possible to position the cursor on top of the pads inserted for standout mode. Instead, it is necessary to erase standout mode using delete and insert line.

The terminal that is unable to correctly transmit the ESC (escape) or **Ctrl-C** characters has **xsb**, indicating that the **F1** key is used for ESC and **F2** for **Ctrl-C**.

Other specific terminal problems can be corrected by adding more capabilities of the form **xx**.

2.3.59.15 Similar Terminals

If two terminals are very similar, one can be defined as being just like the other with certain exceptions. The string capability **use** can be given with the name of the similar terminal. The capabilities given before **use** override those in the terminal type invoked by **use**. A capability can be cancelled by placing **xx@** to the left of the capability definition, where **xx** is the capability. For example, the entry:

```
  term-nl, smkx@, rmkx@, use=term,
```

defines a terminal that does not have the **smkx** or **rmkx** capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

2.3.59.16 Data Base File Names

Compiled **terminfo** descriptions are placed in subdirectories under **/usr/lib/terminfo** in order to avoid performing linear searches through a single directory containing all of the **terminfo** description files. A given description file is stored in **/usr/lib/terminfo/c/name**, where **name** is the name of the terminal, and **c** is the first letter of the terminal name. For example, the compiled description for the terminal **term4-nl** can be found in the file **/usr/lib/terminfo/t/term4-nl** You can create synonyms for the same terminal by making multiple links to the same compiled file. (See the **ln** command in *AIX Operating System Commands Reference* on how to create multiple links to a file.)

Example
The following entry, which describes a terminal, is among the entries in the **terminfo** file.

```
  hft|High Function Terminal,
      cr=^M, cud1=|J, ind=\E[S, bel=^G, il1=\E[L, am, cub1=^H, ed=\E[J,
      el=\E[K, clear=\E[H\E[J, cup=\E[%ip1%d;%p2%dH, cols#80, lines#25,
      dch1=\E[P, dl1=\E[M, home=\E[H,
      ich=\E[%p1%d@, ich1=\E[@, smir=\E[4h, rmir=\E4l,
      bold=\E[1m, rev=\E[7m, blink=\E[5m, invis=\E[8m,  sgr0=\E[0m,
      sgr=\E[%?%p1%t7;%;%?%p2%t4;%;%?%p3%t7;%;%?%p4%t5;%;%?%p6%t1;%;m,
      kcuu1=\E[A, kcud1=\E[B, kcub1=\E[D,
      kcuf1=\E[C, khome=\E[H, kbs=^H, kbtab=\[Z
      cuf1=\E[C, ht=^I, cuu1=\E[A, xon,
      rmul=\E[m, smul=\E[4m, rmso=\E[m, smso=\E[7m,
      kpp=\E[150q, knp=\E[154q,
      kf1=\E[001q, kf2=\E[002q, kf3=\E[003q, kf4=\E[004q,
      kf5=\E[005q, kf6=\E[006q, kf7=\E[007q, kf8=\E[008q,
      kf9=\E[009q, kf10=\E[010q, kf11=\E[011q, kf12=\E[012q,
      bw,    it#8,   ms,
      hpa=\E%i%p1%dG,  ech=\E[%p1dx,
      kdch1=\E[P,   kind=\E[151q,   kich1=\E[139q,   krmir=\E[41,
      knl=^M,   ko=^I,   ktab=^I,   kri=\E[155q,   kend=\E[146q
      cub=\E[%p1%dD, cuf=\E[%p1%dC,  indn=\E[%p1dS, rin=\E[%p1%dT,
      ri=\E[T,   cuu=\E[%p1%dA,  cud=\E[%p1%dB,
      box1=\332\304\277\263    \331\300\302\264\301\303\305,
      box2=\311\315\273\272    \274\310\313\271\312\314\316,
      batt2=md,
```

File

/usr/lib/terminfo/?/* Compiled terminal capability data base.

Related Information
In this book: "curses" in topic 1.2.56, "Terminfo Level Subroutines" in topic 1.2.56.2, "extended curses library" in topic 1.2.74, "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf" in topic 1.2.208, "termdef" in topic 1.2.302, and "TERM" in topic 2.4.26.

The **display** and **tic** commands in *AIX Operating System Commands Reference*.

2.3.60 utmp, wtmp, .ilog

Purpose
Contains user and accounting information.

Synopsis

#include <utmp.h>

Description

When a user logs in successfully, the **login** program writes entries in **/etc/utmp**, the record of users logged into the system, and in **/usr/adm/wtmp** (if it exists), for use in accounting. On invalid login attempts (due to an incorrect login name or password), **login** makes entries in the **/etc/.ilog** file. When you log in as user **root** or **su** and the **/etc/.ilog** file is not empty, you see a message advising you to check the **/etc/.ilog** file for a record of unsuccessful login attempts.

If the Transparent Computing Facility is installed, there is a unique **/etc/utmp** and a unique **/usr/adm/wtmp** for each cluster site (**/etc/utmp** and **/usr/adm/wtmp** are symbolic links into the local file system).

The records in these files follow the **utmp** structure, which is defined in the **utmp.h** header file:

```
    #define UTMP_FILE  "/etc/utmp"
    #define WTMP_FILE  "/usr/adm/wtmp"
    #define ILOG_FILE  "/etc/.ilog"

    #define ut_name  ut_user

    struct utmp
      {
         char ut_user[8];          /* User login name */
         char ut_id[6];            /* id from /etc/inittab */
         char ut_line[12];         /* device name (console, ttyx) */
         pid_t ut_pid;             /* process id */
         short  ut_type;           /* type of entry */
         struct exit_status
           {
            short e_termination;  /* Process termination status */
            short e_exit;         /* Process exit status */
           }
         ut_exit;                  /* The exit status of a process */
                                   /* marked as DEAD_PROCESS. */
         time_t  ut_time;          /* time entry was made */
         char   ut_host[16];       /* host name if remote login */
         long   ut_lsite;          /* reserved */
         datarep_t ut_datarep[4]; /* reserved */
  };



    /* Definitions for ut_type */

    #define EMPTY           0
    #define RUN_LVL         1
    #define BOOT_TIME       2
    #define OLD_TIME        3
```

```
#define NEW_TIME       4
#define INIT_PROCESS   5     /* Process spawned by "init" */
#define LOGIN_PROCESS  6     /* A "getty" process waiting for login */
#define USER_PROCESS   7     /* A user process */
#define DEAD_PROCESS   8
#define ACCOUNTING      9
#define UTMAXTYPE ACCOUNTING  /* Largest legal value of ut_type */

/* Special strings or formats used in the "ut_line" field when */
/* accounting for something other than a process. */
/* No string for the ut_line field can be more than 11 chars + */
/* a NULL in length. */

#define RUNLVL_MSG     "run-level ?"
#define BOOT_MSG       "system boot"
#define OTIME_MSG      "old time"
#define NTIME_MSG      "new time"
```

Files

/etc/utmp	Record of users logged into the system
/usr/adm/wtmp	Accounting information
/etc/.ilog	Record of invalid logins.

The **ut_datarep** field has the value **DR_LOCAL_FORMAT**. This value encodes the CPU byte ordering for the machine on which the entry was created. This value is different for AIX/370 and AIX PS/2.

Related Information
The **login**, **who**, and **write** commands in *AIX Operating System Commands Reference*.

2.4 Chapter 4. Miscellaneous Facilities

Subtopics

2.4.1 About This Chapter

This chapter describes miscellaneous facilities, such as macro packages and character set tables.

2.4.2 ascii

Purpose
Maps the ASCII character set.

Synopsis
cat /usr/pub/ascii

Description
ASCII is a map of the ASCII character set that gives both the octal and hexadecimal equivalents for each character. This file can be printed as needed.

Note: This is neither the PC ASCII nor the AIX ASCII character set. See "data stream" in topic 2.4.3 for information about these character sets.

The contents of this file are:

Figure 4-1. Octal ASCII Character Set

Figure 4-2. Hexadecimal ASCII Character Set

File
/usr/pub/ascii

2.4.3 data stream

Purpose
Defines the data stream that an HFT virtual terminal uses in KSR mode.

Description
AIX is capable of addressing 1024 distinct displayable characters. To designate these characters using 8-bit bytes, a code page convention is used. Each ***code page*** is an ordered set of up to 256 characters, which are called ***code points***. The first 32 code points of each code page are reserved for control codes and are the same for all code pages. The control codes do not have graphic representations, so each code page can have a maximum of 224 distinct graphic characters.

The remaining characters are included in a code page called P0. Two additional code pages called USER1 and USER2 are provided for user-defined symbols.

Code points in the range 32 to 127 (0x20 to 0x7F) of code page P0 represent the standard 7-bit US ASCII graphic symbols. P0 code points 128 to 255 (0x80 to 0xFF).

The following code page map shows the predefined graphic display symbols and their code point values in code page P0.

Subtopics
2.4.3.1 Hardware limitation
2.4.3.2 Nonspacing Characters
2.4.3.3 Controls

2.4.3.1 Hardware limitation

You will not be able to display all the symbols described in this section on the VGA adapter if you have changed from the standard software character mode to the optional hardware character mode. This can be done with either the **display** command or with a **change font** order to the **hft** device driver.

Figure 4-3. Code Page P0

2.4.3.2 Nonspacing Characters

For convenience when typing diacritical (accented) characters, a nonspacing or "dead" character facility is provided. A ***nonspacing character sequence*** is a two-key sequence consisting of one of the 13 diacritics followed by an alphabetic character or a space. The virtual terminal subsystem converts this two-key sequence into a single code point that may have a single-shift prefix. The resulting character is the alphabetic character with the specified diacritic mark. A diacritic followed by a space translates to the diacritic character itself. The 13 valid diacritics are:

If a nonspacing character and the following character do not combine to form a diacritical character in the set of predefined graphic symbols, then the diacritic is treated as a separate character code. For example, **~Q** is treated as two characters, **~** and **Q**.

Note that nonspacing characters apply only to keyboard input and are not a feature of the data stream used by applications. Also, a diacritic must be explicitly designated as being nonspacing in the keyboard mapping for this facility to operate. None of the keys on the standard U.S. keyboard mapping are defined to be nonspacing characters. However, nonspacing characters can be defined. See "Set Keyboard Map (HFSKBD)" in topic 2.5.11.8.5 for details.

2.4.3.3 Controls

Two types of controls are valid in a character stream data:

Single-byte controls (also called ***control characters*** and ***control codes***), which have character values from 0 to 31 (0x00 to 0x1F)

Multi-byte controls, which are also called escape sequences an control sequences.

Subtopics
2.4.3.3.1 Single-Byte Controls
2.4.3.3.2 Multi-Byte Controls

2.4.3.3.1 Single-Byte Controls

The following list shows the single-byte controls and their interpretation in KSR coded data. A line introducing each control gives its mnemonic, its code value, and its function.

NUL, 0x00, (Null) has no terminal function

SOH, 0x01, (Start of Header) has no terminal function

STX, 0x02, (Start of Text) has no terminal function

ETX, 0x03, (End of Text) has no terminal function

EOT, 0x04, (End of Transmission) has no terminal function

ENQ, 0x05, (Enquiry) has no terminal function

ACK, 0x06, (Acknowledge) has no terminal function

BEL, 0x07, (Bell) causes an audible alarm to sound

BS, 0x08, (Backspace) moves the cursor position to the left on column, unless the cursor is at the left boundary of the presentation space. In that case, the cursor position does not change.

HT, 0x09, (Horizontal Tab) moves the cursor position forward to th next tab stop. If the cursor is already in the last column of a line, then the cursor position does not change. Note that the CHT (cursor horizontal tab) multibyte control performs a similar operation, but also performs line wrapping.

LF, 0x0A, (Line Feed) if the LNM mode is reset, the line feed move the cursor position down one line. If the LNM mode is set (default), the line feed is treated as a NEL and moves the cursor position to the first position of the next line. In either case, if the cursor is already on the last line of the PS, the PS lines scroll up one line. The top line of the PS disappears and a blank line is inserted as the new bottom line.

VT, 0x0B, (Vertical Tab) moves the cursor position down to the nex line that is defined as a vertical tab stop. Tabs stops are always set at the first and last lines of the PS. If the cursor was already on the last line of the PS and HFWRAP mode is not set, the cursor stays on the last line in the PS. If HFWRAP mode is set, the cursor moves to the top line in the PS. The column position does not change in any case.

FF, 0x0C, (Form Feed) treated as a line end

CR, 0x0D, (Carriage Return) if the CNM mode is reset (default), th carriage return moves the cursor position to the first character of the line indicated by the cursor. If the CNM mode is set, the carriage return is treated as an NEL and causes the cursor position to move to the first position of the next line. In this case, if the cursor is already on the last line of the PS, the PS lines scroll up one line. The top line of the PS disappears and a blank line is inserted as the new bottom line.

SO, 0x0E, (Shift Out) maps the subsequently received graphic codes t

display symbols according to the active G1 character set. See "display symbols" in topic 2.4.4 for a list of the display symbols.

SI, 0x0F, (Shift In) maps the subsequently received graphic codes t display symbols according to the active G0 character set. See "display symbols" in topic 2.4.4 for a list of the display symbols.

DLE, 0x10, (Data Link Escape) has no terminal function

DC1, 0x11, (Device Control 1) has no terminal function when output

DC2, 0x12, (Device Control 2) has no terminal function

DC3, 0x13, (Device Control 3) has no terminal function when output

DC4, 0x14, (Device Control 4) has no terminal function

NAK, 0x15, (Negative Acknowledgment) has no terminal function

SYN, 0x16, (Synchronous) has no terminal function

ETB, 0x17, (End of Block) has no terminal function

CAN, 0x18, (Cancel) has no terminal function

EM, 0x19, (End of Medium) has no terminal function

SUB, 0x1A, (Substitute) has no terminal function

ESC, 0x1B, (Escape) defines the beginning of a multibyte contro sequence as defined in "Multi-Byte Controls" in topic 2.4.3.3.2.

DEL, 0x7F, (Delete) has no terminal function

2.4.3.3.2 Multi-Byte Controls

This section defines the code points and effects on the virtual terminal for multibyte control sequences that are recognized in KSR mode. All of them begin with the ESC code (0x1B) followed by a **[** (0x5B) and include all subsequent bytes up to and including the first code in the range 0x40--0x7F. Any multibyte control sequences not defined below are ignored. Invalid sequences return an error Device Status Report to the program. Multi-byte control sequences of more than 16 codes are considered invalid on receipt of the 17th code. The next code is not considered a part of that sequence. Also, numeric parameters in control sequences contain no more than 3 digits. The numeric value of the parameter may be incorrect if more than three digits are used, and the numeric value never exceeds 255.

Controls effect a virtual terminal's presentation space (PS) and its related cursor (pointer into the PS). The presentation space is a logical array of display symbols, **N** columns by **M** lines.

The following list gives the valid multibyte control code sequences. A line introducing each control gives its mnemonic, its code sequence, and its function. The code sequence is shown in terms of ASCII characters. For example, the sequence **ESC A** represents two codes with a value of 0x1B41.

CBT ESC [**PN** Z Cursor Back Tab

Moves the cursor back the number of horizontal tab stops specified by **PN**. Tab stops are always set at the first and last columns of each line. If the cursor is already in the first column of a line and HFWRAP mode is set, the cursor moves to the last column. If AUTONL is also set, the cursor moves to the last column of the previous line. In this case, if the cursor is already on the first row of the PS, it moves to the last row.

CHA ESC [**PN** G Cursor Horizontal Absolute

Moves the cursor to the column specified by **PN**, unless the column exceeds the PS width. If the column exceeds the PS width, the cursor moves to the PS column farthest to the right.

CHT ESC [**PN** I Cursor Horizontal Tab

Moves the cursor position forward to the **PN**(th) following tab stop. If the cursor is already in the last column of a line and HFWRAP mode is set, then the cursor returns to the first column of the line. If AUTONL mode is also set, then the cursor moves to the first column of the next line. In this case, if the cursor is already on the last line of the PS, then the cursor moves to the first column of the first line. Note that the HT (horizontal tab) single-byte control does not cause wrapping to occur.

CTC ESC [**PS** W Cursor Tab Stop Control

0 Set a horizontal tab at cursor.
1 Set a vertical tab at cursor.
2 Clear a horizontal tab at cursor.
3 Clear a vertical tab at cursor.
4 Clear all horizontal tabs on line.
5 Clear all horizontal tabs.

6 Clear all vertical tabs.

Sets or clears one or more tabulation stops according to the parameter specified. Tab stops on the first or last column cannot be cleared. When horizontal tab stops are set or cleared, the number of lines affected is all (if Tabulation Stop Mode is set) or one (if Tabulation Stop Mode is reset). This control does not change the position of characters already in the presentation space.

CNL ESC [**PN** E Cursor Next Line

Moves the cursor down the number of lines specified by **PN**, and over to the first position of that line. If the cursor was already on the bottom PS line and HFWRAP mode is not set, it is positioned at the beginning of that line. If HFWRAP mode is set, the cursor wraps from the bottom line to the top PS line.

CPL ESC [**PN** F Cursor Preceding Line

Moves the cursor back the number of lines specified by **PN**, and over to the first position of that line. If the cursor was already on the top PS line and HFWRAP mode is not set, the cursor is positioned at the beginning of that line. If HFWRAP mode is set, the cursor wraps from the top line to the bottom line of the PS.

CPR ESC [**PN** ; **PN** R Cursor Position Report

This is the report that is returned when you issue the Device Status Report Request (see DSR control). The first numeric parameter is the line number, and the second is the column. Line and column values are sent to the application as information. You do not normally send this report to the virtual terminal, but if you do, it is treated as a CUP control.

CUB ESC [**PN** D Cursor Backward

Moves the cursor backward on the line the specified number of columns. If this cursor movement exceeds the left PS boundary and HFWRAP mode is not set, the cursor stops at the leftmost PS position. If HFWRAP mode is set, the cursor wraps from the leftmost column to the rightmost column of the preceding PS line. In HFWRAP mode the cursor also wraps from the home to the rightmost bottom position of the PS.

CUD ESC [**PN** B Cursor Down

Moves the cursor down the number of lines specified by **PN**. If this cursor movement exceeds the bottom PS boundary and HFWRAP mode is not set, the cursor stops on the last PS line. If HFWRAP mode is set, the cursor wraps from the bottom line to the top line of the PS.

CUF ESC [**PN** C Cursor Forward

Moves the cursor forward on the line the specified number of columns. If this cursor movement exceeds the right PS boundary and HFWRAP mode is not set, the cursor stops at the rightmost PS position. If HFWRAP mode is set, the cursor wraps from the rightmost column to the leftmost column of the following line in the PS. In HFWRAP mode, the cursor also wraps from rightmost bottom position to the home position of the PS.

CUP ESC [**PN** ; **PN** H Cursor Position

Moves the cursor to the line specified by the first parameter, and to the column specified by the second parameter. If this movement crosses a PS boundary, the cursor stops at the PS boundary.

CUU ESC [**PN** A Cursor up

Moves the cursor up the specified number of lines. If this cursor movement exceeds the top PS boundary and HFWRAP mode is not set, the cursor stops on the first PS line. If HFWRAP mode is set, the cursor wraps from the top line to the bottom line in the PS.

CVT ESC [**PN** Y Cursor Vertical Tab

Moves the cursor down the number of vertical tab stops specified. Tab stops are assumed at the top and bottom PS lines. If there are not enough vertical tab stops in the PS and HFWRAP mode is not set, the cursor stops on the last line in the PS. If HFWRAP mode is set, the cursor wraps from the bottom line to the top line of the PS.

DCH ESC [**PN** P Delete Character

Deletes the cursor character and the following **PN**-1 characters on the line indicated by the cursor. The characters following the deleted characters on the line overlay the deleted character positions. The line is cleared from the end of the line to the edge of the presentation space. If the number of characters to be deleted exceeds the number of columns from the cursor to the PS right boundary, then all the characters from the cursor to the PS boundary are replaced with empty spaces and a DSR control sequence identifying an error is returned to the application.

DL ESC [**PN** M Delete Line

Deletes the line and the **PN**-1 following lines in the PS. The lines following the deleted lines are scrolled up **PN** lines and **PN** blanks lines are placed at the bottom of the PS. If there are less than **PN** lines from the line indicated by the cursor to the bottom of the PS, the line indicated by the cursor and all the following PS lines are replaced with empty lines.

DSR ESC [**PN** n Device Status Report Request

6 Request Cursor Position Report

13 Error Report

A request cursor position report (CPR) sends a cursor position report from the virtual terminal to the application. An error report is sent from the virtual terminal to the application when the virtual terminal receives an invalid control sequence. Error reports are private reports which conform to the ANSI standard for private parameters.

DMI ESC ' (left quote) Disable Manual Inpu

This control, when received in an output data stream, causes keyboard input to this terminal to be ignored. This control is ignored when received from the keyboard.

EMI ESC b Enable Manual Inpu

This control, when received in an output data stream, restarts keyboard input recognition and buffering if previously disabled with a DMI multibyte control. This control is ignored when received from the keyboard.

EA ESC [0 O Erase to End of Are

ESC [1 O Erase from Start of Area

ESC [2 O Erase All of Area.

This control is treated like an EL control sequence.

ED ESC [0 J Erase to End of Displa

ESC [1 J Erase from Start of Display

ESC [2 J Erase All of Display.

Erases certain characters within the PS. Erased characters are replaced with empty spaces. Erase to end of display erases the character indicated by the cursor and all following characters in the PS. Erase from start of display erases the first character of first line and the following characters up to and including the character indicated by the cursor. Erase all of display erases all the characters on the PS.

EF ESC [0 N Erase to End of Fiel

ESC [1 N Erase from Start of Field

ESC [2 N Erase All of Field.

Erases certain characters between horizontal tab stops. Erased characters are replaced with empty spaces. Erase to end of field erases the character indicated by the cursor and all following characters before the next tab stop. Erase from start of field erases the character at the tab stop preceding the cursor an the following characters up to and including the character indicated by the cursor. Erase all of field erases the character at the tab stop preceding the cursor, and the following characters up to and including the character at the tab stop following the cursor. Tab stops are assumed at the first and last columns of the PS when executing this control.

EL ESC [0 K Erase to End of Lin

ESC [1 K Erase from Start of Line

ESC [2 K Erase All of Line.

Erases certain characters within a line. Erased characters are replaced with empty spaces. Erase to end of line erases the character indicated by the cursor and all following characters on the line. Erase from start of line erases the first character of first line and the following characters up to and including the character indicated by the cursor. Erase all of line erases all the characters on the line.

ECH ESC [**PN** X Erase Character

Erases the character indicated by the cursor and the following **PN**-1 characters on that line. Erased characters are replaced with empty spaces. If there are less than **PN** characters from the cursor to the PS right boundary, then the character indicated by the cursor and all the following characters on the line are replaced empty spaces.

HTS ESC H Horizontal Tab Sto

Sets a horizontal tab stop at the current horizontal position. If TSM is set, then the tab stop applies only to this line. If TSM is reset, then the tab stop applies to all PS lines. This control does not change the positioning of characters already in the presentation space.

HVP ESC [**PN** ; **PN** f Horizontal and Vertical Position

Moves the cursor to the line specified by the first parameter, and to the column specified by the second parameter. If this movement would cross a PS boundary, the cursor stops at the current PS boundary.

ICH ESC [**PN** @ Insert Character

Inserts **PN** empty spaces before the character indicated by the cursor. The string of characters starting with the character indicated by the cursor and ending with last character of the line are shifted **PN** columns to the right. Characters shifted past the PS right boundary are lost. The cursor does not move.

IL ESC [**PN** L Insert Line

Inserts **PN** empty lines before the line indicated by the cursor. The line indicated by the cursor is scrolled down. The cursor position on the screen is not affected.

IND ESC D Inde

Moves cursor down one line. If the cursor was already on the bottom line of the PS, then the top line is lost, the other lines move up one line, and a blank line becomes the new bottom line.

NEL ESC E Next Lin

Moves the cursor to the first position of the following line. If the cursor was already on the bottom line of the PS, then the top line is lost, the other lines move up one, and a blank line becomes the new bottom line.

KSI ESC [**PS** p Keyboard Status Information

The virtual terminal generates this control whenever **HFHOSTS** and **HFXLATKBD** are set and the status of the keyboard changes. Each selective parameter is the character-coded decimal value of a keyboard status byte. For example, if the keyboard has two status bytes, the control sequence is **ESC [xxx;yyy p**, where **xxx** is the value of the high-order byte and **yyy** is the value of the low-order byte. This is a private control that conforms to the ANSI standards for private control sequences. The virtual terminal display handler ignores this sequence whether it is received from the application or echoed.

PFK ESC [**PN** q PF Key Report

The control sequence is sent by the virtual terminal to the application when a program function key (PFK) code is received from the keyboard. The parameter **PN** is a PF key number from 1 to 255. This is a private control that conforms to the ANSI standards for private control sequences. This sequence is ignored by the virtual terminal display handler whether received from the application or echoed.

RCP ESC [u Restore Cursor Positio

Moves the cursor to the position saved by the last SCP control. If no SCP has been received, then the cursor position is set to the first character of the first line. This is a private control that conforms to the ANSI standards for private controls. This control has no terminal function when received from the keyboard.

RI ESC L Reverse Inde

Moves the cursor up one line, unless the cursor is already on the PS top line. In that case, if HFWRAP mode is not set, then the cursor does not move. If HFWRAP mode is set, the cursor moves to the bottom line of the PS. The column position does not change.

RIS ESC c Reset to Initial Stat

Resets the virtual terminal to the state of a newly-opened virtual terminal: erases all PS data, places the cursor at the home position, resets graphic rendition to normal, resets subscripting and superscripting, and sets tab stops, modes, keyboard map, character maps and echo maps to their default values.

RM ESC [**PS** l Reset Mode

20 LNM - Line Feed - New Line Mode
4 IRM - Insert Mode
12 SRM - Send Receive Mode (set ECHO off)
18 TSM - Tabulation Stop Mode
?21 CNM - Carriage Return - New Line Mode
?7 AUTONL - Wrap character to following line when end of current line reached

Resets the modes specified in the parameter string. Multiple parameters must be separated by semicolons. The modes that can be reset are listed above with the appropriate parameter code. All other mode parameters are ignored.

TSM mode determines whether horizontal tabs apply identically to all line (TSM reset) or uniquely to each line on which they are set (TSM set).

SCP ESC [s Save Cursor Positio

Saves the current cursor position. Any previously saved cursor position is lost. The cursor can be restored to this position with an RCP control. This is a private control that conforms to the ANSI standards for private controls. This control has no terminal function when received from the keyboard.

SD ESC [**PN** T Scroll Down

Moves all the PS lines down **PN** lines. The bottom **PN** lines are lost, and **PN** empty lines are put at the top of the presentation space. Physical cursor position does not change due to the scroll.

SL ESC [**PN** SP @ Scroll Left

Moves all the PS characters **PN** column positions to the left. The characters in the **PN** leftmost PS columns are lost, and empty spaces are put in the rightmost **PN** columns of all lines. Physical cursor position does not change due to the scroll.

SR ESC [**PN** SP A Scroll Right

Moves all the PS characters **PN** column positions to the right. The characters in the **PN** rightmost PS columns are lost, and empty spaces are put in the leftmost **PN** columns of all lines. Physical cursor position does not change due to the scroll.

SU ESC [**PN** S Scroll Up

Moves all the PS lines up **PN** lines. The top **PN** lines are lost, and **PN** empty lines are put at the bottom of the presentation space. The physical cursor position does not change due to the scroll.

SGR ESC [**PS** m Set Graphic Rendition

0 Normal (none of attributes 1-9)
1 Bold or Bright
4 Underscore
5 Slow Blink
7 Negative (reverse image)
8 Cancelled On (invisible: set to background color)
10 Primary Font
11 First Alternate Font
12 Second Alternate Font
13 Third Alternate Font
14 Fourth Alternate Font
15 Fifth Alternate Font
16 Sixth Alternate Font
17 Seventh Alternate Font
30 Color palette entry 0 foreground
31 Color palette entry 1 foreground
32 Color palette entry 2 foreground
33 Color palette entry 3 foreground
34 Color palette entry 4 foreground
35 Color palette entry 5 foreground
36 Color palette entry 6 foreground
37 Color palette entry 7 foreground
40 Color palette entry 0 background
41 Color palette entry 1 background
42 Color palette entry 2 background
43 Color palette entry 3 background
44 Color palette entry 4 background
45 Color palette entry 5 background
46 Color palette entry 6 background
47 Color palette entry 7 background
90 Color palette entry 8 foreground

91 Color palette entry 9 foreground
92 Color palette entry 10 foreground
93 Color palette entry 11 foreground
94 Color palette entry 12 foreground
95 Color palette entry 13 foreground
96 Color palette entry 14 foreground
97 Color palette entry 15 foreground
100 Color palette entry 8 background
101 Color palette entry 9 background
102 Color palette entry 10 background
103 Color palette entry 11 background
104 Color palette entry 12 background
105 Color palette entry 13 background
106 Color palette entry 14 background
107 Color palette entry 15 background.

Causes the next characters received in the data stream or from the keyboard to have the display attributes specified by the parameter string. Any parameter not listed above is ignored.

The attributes corresponding to parameters 1 through 9 are cumulative. For example, specifying **underscore** and then specifying **blink** causes following characters to be underscored and blink. To reset one of these attributes, specify **normal** and then reinstate the desired parameters. Multiple parameters are processed in the order listed.

Whether the characters really have the requested attributes on the display depends on the capabilities of the physical display device used by the virtual terminal. The VGA adapter of the PS/2 does not implement all of the defined capabilities. See "Change Fonts" in topic 2.5.11.7.3 in the **hft** section of this manual for more information on this topic.

You can obtain the attributes of any portion of the screen by issuing the **query presentation space** command which is also defined in the **hft** section of this manual under "Query Presentation Space Command" in topic 2.5.11.5.6.

Characters that cannot be displayed do not exist in the system.

SG0A ESC (f Set G0 Character Se

SG0B ESC , f Set G0 Character Set (Alternate form)

: Unique One (User-defined)
; Unique Two (User-defined)
< P0 (Display Symbols 32-255)
= P1 (Display Symbols 256-479)
> P2 (Display Symbols 480-703)
? User1 (Display Symbols 704-927)
@ User2 (Display Symbols 928-1023)

Designates the set of characters to use as the G0 set when the G0 set is invoked by SI. The default G0 set is the 224-character code page P0. Unique One and Unique Two may have unique definitions for each virtual terminal. When a virtual terminal is opened, these two sets are equivalent to **<**. See "Set User-Defined Character Set" in topic 2.5.11.8.3 about defining Unique One and Unique Two.

SG1A ESC) f Set G1 Character Se

SG1B ESC - f Set G1 Character Set (Alternate)

:	Unique One (User-defined)	
;	Unique Two (User-defined)	
<	P0	(Display Symbols 32-255)
=	P1	(Display Symbols 256-479)
>	P2	(Display Symbols 480-703)
?	User1	(Display Symbols 704-927)
@	User2	(Display Symbols 928-1023)

Designates the set of characters to use as the G1 set when the G1 set is invoked by SO. The default G1 set is the 224-character code page P0. Unique One and Unique Two may have unique definitions for each virtual terminal. When a virtual terminal is opened, these two sets are equivalent to **<**. See "Set User-Defined Character Set" in topic 2.5.11.8.3 about defining Unique One and Unique Two.

SM ESC [**PS** h Set Mode

2 0	LNM - Line Feed - New Line Mode (default = 1)
4	IRM - Insert Replace Mode (default = 0)
1 2	SRM - Send Receive Mode (set echo off) (default = 0)
1 8	TSM - Tabulation Stop Mode (default = 0)
? 2 1	CNM - Carriage Return - New Line Mode (default = 0)
? 7	AUTONL - Wrap to next line when end of line reached (default = 1)

Sets the modes specified in the parameter string. Multiple parameters must be separated by semicolons. The modes that can be set are listed above with the appropriate parameter code. All other mode parameters are ignored.

SRM mode affects translated keyboard input handling. If SRM mode is set, translated keyboard input is never echoed by the virtual terminal, but is immediately returned to the application.

TSM mode determines whether horizontal tabs apply to all lines identically (TSM reset) or if horizontal tabs apply uniquely to each line on which they are set (TSM set).

TBC ESC [**PS** g Tabulation Clear

0	Horizontal tab at cursor column
1	Vertical tab at line indicated by the cursor
2	Horizontal tabs on line
3	Horizontal tabs in presentation space
4	Vertical tabs in presentation space.

Clears tabulation stops specified by the parameters. Horizontal tab changes affect only the line indicated by the cursor if TSM is set, and horizontal tab changes affect all lines if TSM is reset. Any parameters not listed above are ignored. This control does not change the positioning of characters already in the presentation space.

VTA ESC [r Virtual Terminal Addressabilit

This private control sequence precedes a binary header and associated data that provide status information on the IBM 5081 Display Adapter.

VTD ESC [x Virtual Terminal Dat

This private control sequence precedes a binary header and associated data. The block of data can be in formats other than character-coded data, such as binary format.

VTL ESC [y Virtual Terminal Device Inpu

This private control sequence precedes binary format input data from a mouse, tablet, LPFK, or valuator device. See "Input Device Report" in topic 2.5.11.3.1 for details about how this control sequence is used.

VTR ESC [w Virtual Terminal Raw Keyboard Inpu

This private control sequence precedes "raw" (untranslated) keyboard input data, which is in a binary format.

VTS ESC I Vertical Tab Sto

Sets a vertical tab stop at the line indicated by the cursor. This control does not change the positioning of characters already in the presentation space.

File

/usr/pub/charset.ibm Contains AIX character set.

Related Information

In this book: "display symbols" in topic 2.4.4 and "hft" in topic 2.5.11.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

The **display** command in *AIX Operating System Commands Reference*.

2.4.4 display symbols

Purpose
Defines the set of US and European character symbols that can be displayed on an HFT display device in KSR mode.

Description

Each character code passed in KSR data is translated into one of 1024 10-bit display symbol codes. Codes 0 through 703 (0x2bf) are predefined to be common across all virtual terminals. Codes 704 (0x2c0) through 1023 (0x3ff) are reserved for user-defined extensions to the display symbol set. Display symbols 0 through 31 (0x1f) represent control functions and have no graphic representations.

Code page P0 contains all of the predefined characters. In addition to the predefined code page P0, you can define two code pages called Unique One and Unique Two. See "fonts" in topic 2.3.19, "data stream" in topic 2.4.3, and "Reconfigure (HFRCONF)" in topic 2.5.11.8.2 for information you need to define such character sets.

The columns of the following tables represent:

Font Position
The position of the graphic display symbol within the font definition.

Code Page/Code Point
The code page of the symbol and the offset within that code page.

char String
The internal hexadecimal representation as a string of type **char**.

NLchar Value
The value of the **NLchar** data type that corresponds to the character.

NCesc Esc Seq
The ASCII character or escape sequence that corresponds to the character after being translated by the **NCesc** macro. See "conv" in topic 1.2.50 and "NLescstr, NLunescstr, NLflatstr" in topic 1.2.189 for related information.

Subtopics
2.4.4.1 Hardware limitation

2.4.4.1 Hardware limitation

You will not be able to display all the symbols described in this section on the VGA adapter if you have changed from the standard software character mode to the optional hardware character mode. This can be done with either the **display** command or with a **change font** order to the **hft** device driver.

The table begins at font position 32 because the first 32 positions are reserved for the single-byte controls.

File

/usr/pub/ibmcharset Contains PS/2 character set.

Related Information
In this book: "conv" in topic 1.2.50, "NLchar" in topic 1.2.188, "NLescstr, NLunescstr, NLflatstr" in topic 1.2.189, "fonts" in topic 2.3.19, "data stream" in topic 2.4.3, "hft" in topic 2.5.11, and "keyboard" in topic 2.5.13.

2.4.5 ebcdic

Purpose
Maps the EBCDIC character set.

Synopsis
cat /usr/pub/ebcdic

Description
In the following table columns correspond to the high-order hexadecimal digits and rows correspond to low-order hexadecimal digits. The cells contain equivalent hexadecimal ASCII values, the symbols, and mnemonics common to EBCDIC and ASCII. Exceptions are flagged in Figure 4-5.

Figure 4-4. EBCDIC Character Set

	EBCDIC		**ASCII**	
(1)	13	tm	13	dc3
(2)	1C	ifs	1C	fs
(3)	1D	igs	1D	gs
(4)	1E	irs	1E	rs
(5)	1F	ius	1F	us
(6)	4F	or	7C	or
(7)	5F	not	7E	¬
(8)	9A		5E	^

Figure 4-5. EBCDIC and ASCII Character Set Exceptions

Note: The information provided in these tables describe only the EBCDIC character set which is used with the C locale. Other extended EBCDIC character sets are used for other locales. See the **iconv** command in the *AIX Commands Reference* for more information.

File
/usr/pub/ebcdic

Related Information
The **dd** command in *AIX Operating System Commands Reference*. The **iconv**, **axeb**, and **ebxa** commands in the *AIX Commands Reference*.

2.4.6 environment

Purpose
Describes the user environment.

Synopsis

Subtopics
2.4.6.1 Basic Environment
2.4.6.2 International Character Support Environment
2.4.6.3 The Basic Environment

2.4.6.1 Basic Environment

HOME=path name of home directory
PATH=directory search sequence
TERM=terminal type
TZ=time zone information
LOGNAME=user's login name
USER=user's login name
LOGTTY=user's logged-in terminal line
SHELL=default shell (**sh** only)
CWD=current working directory
MAIL=user's mailbox

2.4.6.2 International Character Support Environment

NLIN=path name of the EBCDIC-to-ASCII translation table
NLOUT=path name of the ASCII-to-EBCDIC translation table

LANG=names entire **locale**
LC_COLLATE=ordering of characters
LC_CTYPE=character examination functions
LC_MONETARY=monetary values
LC_NUMERIC=type of decimal point
LC_TIME=time conversion
LC_MESSAGE=language of error messages

The following environment variables are understood only by executables generated under the AIX 1.2 or earlier environment. The same functionality can be achieved by using the environment variables used by **setlocale**.

NLFILE=path name of environment file
NLCTAB=path name of collating tables
NLLANG=language name

NLCURSYM=currency symbol
NLNUMSEP=triad and decimal separators

NLLDAY=long day names
NLLMONTH=long month names
NLSDAY=short day names
NLSMONTH=short month names
NLTMISC=miscellaneous time strings
NLTSTRS=relative time names
NLTUNITS=time unit names

NLDATE=short date format
NLLDATE=long date format
NLTIME=time format

Description
When a new process begins, the **exec** system call makes an array of strings available that have the form **name=value**. This array of strings is called the ***environment***. Each **name** defined by one of the strings is called an ***environment variable*** or ***shell variable***.

When using the **sh** command interpreter, additional names can be placed in the environment with the **export** or **env** command, or by adding a **name=value** prefix to any other command. See the **sh** command in *AIX Operating System Commands Reference* for more information about setting environment variables with shell commands.

When using the **csh** command interpreter, additional names can be placed in the environment with the **setenv** command. See the **csh** command in *AIX Operating System Commands Reference* for more information about setting environment variables with shell commands.

Within a program, the **getenv** subroutine can be used to search the environment for the value of a given variable. The **exec** system call allows the entire environment to be set at one time, usually for a newly started child process.

When creating new environment variables, ensure that their names do not

conflict with those of standard variables used by the shell and other programs, such as **MAIL**, **PS1**, **PS2**, and **IFS**.

2.4.6.3 The Basic Environment

When you log in, a number of environment variables are automatically set by the system before running your login profile, **.profile**. These variables make up the *basic environment*:

CWD Current working directory.

HOME The full path name of the user's login or home directory. The **login** program sets this to the name specified in the **/etc/passwd** file.

LOGNAME User's login name.

LOGTTY User's logged-in terminal.

MAIL User's mailbox.

NLSPATH Contains a sequence of templates which **catopen** uses when attempting to locate message catalogs. Each template consists of an optional prefix, one or more substitution fields, a filename and an optional suffix.

For example:

```
NLSPATH="/system/nlslib/%N.cat"
```

indicates that **catopen** should look for all message catalogs in the directory **/system/nlslib**, where the catalog name should be constructed from the **name** parameter passed to **catopen**, %N, with the suffix **.cat**.

Substitution fields consist of a % (percent) symbol, followed by a single letter keyword. The following keywords are currently defined:

%N The value of the **name** parameter passed to **catopen**.

%L The value of **LANG**.

%l The **language** element from **LANG**.

%t The **territory** element from **LANG**.

%c The **codeset** element from **LANG**.

%% A single % character.

An empty string is substituted if the specified value is not currently defined. The separators _ and . are not included in **%t** and **%c** substitutions.

Templates defined in NLSPATH are separated by colons (:). A leading colon or two adjacent colons (::) is equivalent to specifying **%N**. For example:

```
NLSPATH=":%N.cat:/nlslib/%L/%N.cat"
```

indicates to **catopen** that it should look for the requested message catalog in **name**, **name.cat**, and **nlslib/$LANG/name.cat**.

PATH The sequence of directories that commands such as **sh**, **time**, **nice**, and **nohup** search when looking for a command whose path name is incomplete. The directory names are separated by colons. **PATH** is set by the system login profile, **/etc/profile**.

SHELL User's shell (**sh** only).

TERM The type of terminal for which output is to be prepared. Commands such as **mm** and **tplot** use this information to manipulate special capabilities, if any, of that terminal. The **curses**, **extended curses**, and **terminfo** subroutines also use the value of **TERM**. **TERM** is set by the **getty** command to a value defined in **/etc/ports**.

TZ Time zone information. **TZ** is set in the system environment file, **/etc/environment**.

The value of **TZ** has the following form (spaces inserted for clarity):

std offset dst offset, rule

The expanded format is as follows:

stdoffset[dst[offset][,start[/time],end[/time]]]

Where:

std and *dst* Three or more bytes that are the designation for the standard (**std**) or summer (**dst**) time zone. Only **std** is required; if **dst** is missing, then summer time does not apply in this locale. Upper- and lowercase letters are explicitly allowed. Any characters except a leading colon (:), digits, comma (,), minus (-), plus (+), and ASCII NULL are allowed.

offset Indicates the value one must add to the local time to arrive at Coordinated Universal Time. The **offset** has the following form:

hh[:mm[:ss]]

The minutes (**mm**) and seconds (**ss**) are optional. The hour (**hh**) shall be required and may be a single digit. The **offset** following **std** shall be required. If no **offset** follows **dst**, summer time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour shall be between 0 and 24, and the minutes (and seconds) - if present - between 0 and 59. Out of range values may cause unpredictable behavior. If preceded by a "-", the time zone shall be east of the Prime Meridian; otherwise it shall be west (which may be indicated by an optional preceding "+").

rule Indicates when to change to and from summer time. The **rule** has the following form:

date/time,date/time

where the first **date** describes when the change from standard to summer time occurs and the second **date** describes when the change back happens. Each **time** field describes when, in current local time, the change to the other time is made.

The format of **date** shall be one of the following:

J*n* The Julian day **n** (1=**n**=365). Leap days shall not be counted. That is, in all years - including leap years - February 28 is day 59 and March 1 is day 60. It is impossible to explicitly refer to the occasional February 29.

n The zero-based Julian day (0=**n**=365). Leap days shall be counted, and it is possible to refer to February 29.

M*m.n.d* The **d**(th) day (0=**d**=6) of week **n** of month **m** of the year (1=**n**=5, 1=**m**=12, where week 5 means "the last **d** day in month **m**" which may occur in either the fourth or the fifth week). Week 1 is the first week in which the **d**(th) day occurs. Day 0 is Sunday.

The **time** has the same format as **offset** except that no leading sign ("-" or "+") shall be allowed. The default, if **time** is not given, shall be 02:00:00.

USER User's login name.

Subtopics
2.4.6.3.1 International Character Support Environment

2.4.6.3.1 International Character Support Environment

A special set of environment variables defines the international character support configuration. These environment variables locate configuration information and tailor input and output forms of dates, times, and monetary sums according to "national" or local requirements. If an environment variable **value** for international character support contains blanks, the value appears in quotes and blanks cannot separate the equals sign from the variable name or the value.

The **NLgetenv** subroutine provides a program with a method to retrieve a value associated with an international character support environment variable.

Note that when the **NLgetenv** subroutine is called, the values returned are associated with internal variables maintained by the **setlocale** subroutine. If you do not call **setlocale** or the front-ends to it (**NLgetctab** and **NLgetfile**), **NLgetenv** returns the hard-coded **C** locale default values. If you do call **setlocale**, **NLgetenv** returns the values of the tokens defined in **/usr/lib/mbcs/$LANG** (the **setlocale** subroutine binds the user's language requirements, as specified by **LANG**, to a program's locale).

The environment variables **NLCTAB** and **NLFILE** are used by the **NLgetctab** and **NLgetfile** subroutines to resolve the environment file path. If you wish to write portable programs, avoid using these routines which are only front-ends to **setlocale** (see "setlocale" in topic 1.2.251 and "nl_langinfo" in topic 1.2.198).

The environment variables are described as follows:

LANG Identifies the user's requirements as an ASCII character string in the form:

```
LANG=language[_territory[.codeset]]
```

Specific language operation is initiated at program start-up by calling the **setlocale** subroutine, binding the user's requirements specified in **LANG** to a program's locale as follows:

```
setlocale (LC_ALL, "");
```

On AIX systems, this form of a **setlocale** call is defined to initialize the program's entire locale from the associated environment variables. **LANG** names the program's entire locale and it provides the necessary defaults if any of the category variables are not set or set to the empty string.

LC_COLLATE Loads in a new collation table that determines the ordering of characters. **LC_COLLATE** affects the behavior of all subroutines which handle character comparisons.

LC_TYPE Loads in a new **ctype** table to be used for handling regular expressions and affects the behavior of all subroutines which handle characters with typing information.

LC_MONETARY Affects the monetary formatting information, such as language, territory and codeset, returned by the **localeconv** subroutine.

LC_NUMERIC Affects the decimal point character for the formatted input/output functions in **scanf** and **printf** and the string conversion functions in **strtod**, as well as the non_monetary information returned by the **localeconv** subroutine.

LC_TIME Affects the time and weekday functions in the **strftime** subroutine.

LC_MESSAGE Specifies the language in which the system error messages are to be displayed.

The following environment variables are understood only by executables generated under the AIX 1.2 environment. The same functionality can be achieved by using the environment variables used by **setlocale**.

NLCTAB The path name of the file containing tables that define the current collating sequence, as produced by the **ctab** command. The default path name is:

```
/etc/nls/ctab/default
```

NLCURSYM The currency symbol name and placement. The default value is:

```
:$:L:
```

NLDATE The environment format string specifying the short form of the date. This format is used by **NLstrtime** when the format **%D** is encountered. The default is:

```
MM/DD/YY
```

NLFILE The path name of a file containing other environment variable definitions for international character support. **NLFILE** cannot be defined within a file that is identified by another **NLFILE** definition. There is no default path name.

NLIN The value of **NLIN** is the path name of the EBCDIC-to-ASCII translation table. The path name may be an absolute path name or a partial path name. The path name may also be the name of a system supplied translation table or a user configured translation table. For a complete list of system supplied tables, see **/usr/lib/nls/nlin**.

NLLANG The environment language label for the set of variables and environment format strings used for language conventions. The default value is:

```
u.s.english
```

NLLDATE The environment format string specifying the long form of the date. This form is used by **NLstrtime** when the formats **%lD** or **%sD** are encountered. The default long date format string is:

```
mon DD, YYYY
```

NLLDAY The full (long) names for the days of the week. The default value is:

Sunday:Monday:Tuesday:Wednesday:Thursday:Friday:Saturday

NLLMONTH The full (long) names for the months of the year. The default value is:

January:February:March:April:May:June:July:\
August:September:October:November:December

NLNUMSEP The numeric triad and decimal separators. The first of the two separators is the triad separator, which is used to separate groups of three digits in decimal values. The default value for **NLNUMSEP** is:

:,:.:

NLOUT The value of **NLOUT** is the path name of the ASCII-to-EBCDIC translation table. The path name may be an absolute path name or a partial path name. The path name may also be the name of a system supplied translation table or a user configured translation table. For a complete list of system supplied tables, see **/usr/lib/nls/nlout**.

NLSDAY The short names of the days of the week. Names should be the same length, and of 5 or fewer characters. The default short name string is:

Sun:Mon:Tue:Wed:Thu:Fri:Sat

NLSMONTH The short names of the months of the year. Names should be the same length, and of 5 or fewer characters. The default value is:

Jan:Feb:Mar:Apr:May:Jun:Jul:Aug:Sep:Oct:Nov:Dec

NLTIME The environment format string specifying the format of the time, that is used by **NLstrtime** when the formats **%T**, **%sT**, or **%r** are encountered. The default time format string is:

hh:mm:ss

NLTMISC Miscellaneous strings needed for input and output of date and time specifications. The default miscellaneous string value is:

at:each:every:on:through:am:pm:zulu

NLTSTRS The relative or informal names needed for input of date and time specifications to the **remind** and **at** commands (see the **remind** and **at** commands in *AIX Operating System Commands Reference*). The default informal time string value is:

now:yesterday:tomorrow:noon:midnight:next:weekdays:weekend:toc

NLTUNITS The singular and plural forms for all names of units of time, used for input of date specifications to the **at** command. The default string value for units of time is:

minute:minutes:hour:hours:day:days:week:weeks:month:months:year:years:min:mins

Files

/etc/environment	Sets the basic environment for all processes.
/etc/profile	Allows variables to be added to the environment by the shell.
$HOME/.profile	Sets the environment for a specific user's needs.
/etc/nls/ctab/default	Sets the international character support environment (Release 1.2 only).

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" in topic 1.2.71, "getenv, NLgetenv" in topic 1.2.94, "NLstrtime" in topic 1.2.194, "NLtmtime" in topic 1.2.195, "termdef" in topic 1.2.302, "passwd" in topic 2.3.44, "profile" in topic 2.3.48, "TERM" in topic 2.4.26, "setlocale" in topic 1.2.251, "nl_langinfo" in topic 1.2.198, and "strftime" in topic 1.2.287.

The **ctab**, **env**, **export**, **login**, and **sh** commands in *AIX Operating System Commands Reference*.

"Introduction to International Character Support" in *Managing the AIX Operating System*.

AIX Guide to Multibyte Character Set (MBCS) Support.

2.4.7 eqnchar

Purpose
Identifies special character definitions for **eqn** and **neqn** formatters.

Synopsis

eqn /usr/pub/eqnchar [**files**] | **troff** [options]
neqn /usr/pub/eqnchar [**files**] | **nroff** [options]

Description
The **eqnchar** file contains **troff** and **nroff** character definitions used to construct special scientific symbols. These definitions are primarily intended to be used with the **eqn** and **neqn** formatters. The **eqnchar** file contains definitions for the following characters:

File
/usr/pub/eqnchar

Related Information
The **eqn**, **nroff**, and **troff** commands in *AIX Operating System Commands Reference*.

2.4.8 fcntl.h

Purpose
Defines file control options.

Synopsis
#include <fcntl.h>

Description
The **fcntl.h** header file defines the values that can be specified for the **cmd** and **arg** parameters of the **fcntl** system call, and for the **oflag** parameter of the **open** system call.

```
/* Flag values accessible to open and fcntl */
/* The first three can only be set by open */

#define O_RDONLY  0
#define O_WRONLY  1
#define O_RDWR    2
#define O_NDELAY  04       /* Non-blocking I/O */
#define O_APPEND  010      /* (0x08) append (writes guaranteed */
                                  /*        at the end) */
#define O_SYNC  0200000    /* Synchronous write */
#define O_REPLSYNC 0400000 /* Synchronous replicated write */

/* Flag values accessible only to fcntl */
#define O_ASYNC  020       /* (0x10) signal pgrp when data ready */
/* Flag values accessible only to open */

#define O_DEFERC 00040     /* (0x0020) delay commits of changes */
#define O_EDELAY 00100     /*Return EAGAIN if block would have occurred*/
#define O_NONBLOCK (O_NDELAY | O_EDELAY)      /* POSIX compatibility */
#define O_CREAT  00400     /* (0x0100) open with create (uses third */
                                 /*                open arg) */
#define O_TRUNC  01000     /* (0x0200) open with truncation */
#define O_EXCL   02000     /* (0x0400) exclusive open */

/* fcntl requests */
#define F_DUPFD  0         /* Duplicate fildes */
#define F_GETFD 1          /* Get fildes flags */
#define F_SETFD 2          /* Set fildes flags */
#define F_GETFL 3          /* Get file flags */
#define F_SETFL 4          /* Set file flags */
#define F_GETLK 5          /* Get file lock */
#define F_SETLK 6          /* Set file lock */
#define F_SETLKW 7         /* Set file lock and wait */
#define F_GETOWN  9        /* get pgrp */
#define F_SETOWN 10        /* set pgrp for SIGIO */


/* file segment locking set data type - information passed to */
/* system by user */

struct flock {
     short   l_type;
     short   l_whence;
     long    l_start;
     long    l_len;      /* len = 0 means until end of file */
     unsigned long   l_sysid;
     pid_t   l_pid;
```

```
};

/* file segment locking types */
#define F_RDLCK 01     /* Read lock */
#define F_WRLCK 02     /* Write lock */
#define F_UNLCK 03     /* Remove lock(s) */
```

File

/usr/include/fcntl.h

Related Information

In this book: "fcntl, flock, lockf" in topic 1.2.78 and "open, openx, creat" in topic 1.2.199.

2.4.9 greek

Purpose
Maps Greek characters.

Synopsis
cat /usr/pub/greek [| **greek -Tterminal**]

Description
The **/usr/pub/greek** file shows the mapping from ASCII characters to the "shift-out" graphics in effect between **SO** and **SI** on TELETYPE Model 37 workstations equipped with an extended (128) character set. These codes are the default Greek characters produced by the **nroff** command. Use the **greek** command to translate these characters for display on other workstations. The file contains:

File
/usr/pub/greek

Related Information
The **300**, **4014**, **450**, **greek**, **hp**, **nroff**, **tc**, and **troff** commands in *AIX Operating System Commands Reference*.

2.4.10 langinfo.h

Purpose
Contains language information constants.

Synopsis

#include <langinfo.h>

Description

This header file contains the constants used to identify langinfo data items (see "nl_langinfo" in topic 1.2.198). The header file **nl_types.h** describes the mode of the constants defined below.

The entries under 'Category' refer to the categories defined in the **setlocale** subroutine.

Constant	**Category**	**Meaning**
D_T_FMT	LC_TIME	string for formatting date and time
D_FMT	LC_TIME	date format string
T_FMT	LC_TIME	time format string
AM_STR	LC_TIME	Ante Meridiem affix
PM_STR	LC_TIME	Post Meridiem affix
DAY_1	LC_TIME	name of the first day of the week (e.g., Sunday)
DAY_2	LC_TIME	name of the second day of the week (e.g., Monday)
DAY_3	LC_TIME	name of the third day of the week (e.g., Tuesday)
DAY_4	LC_TIME	name of the fourth day of the week (e.g., Wednesday)
DAY_5	LC_TIME	name of the fifth day of the week (e.g., Thursday)
DAY_6	LC_TIME	name of the sixth day of the week (e.g., Friday)
DAY_7	LC_TIME	name of the seventh day of the week (e.g., Saturday)
ABDAY_1	LC_TIME	abbreviated name of the first day of the week
ABDAY_2	LC_TIME	abbreviated name of the second day of the week
ABDAY_3	LC_TIME	abbreviated name of the third day of the

		week
ABDAY_4	LC_TIME	abbreviated name of the fourth day of the week
ABDAY_5	LC_TIME	abbreviated name of the fifth day of the week
ABDAY_6	LC_TIME	abbreviated name of the sixth day of the week
ABDAY_7	LC_TIME	abbreviated name of the seventh day of the week
MON_1	LC_TIME	name of the first month in the Gregorian calendar
MON_2	LC_TIME	name of the second month
MON_3	LC_TIME	name of the third month
MON_4	LC_TIME	name of the fourth month
MON_5	LC_TIME	name of the fifth month
MON_6	LC_TIME	name of the sixth month
MON_7	LC_TIME	name of the seventh month
MON_8	LC_TIME	name of the eighth month
MON_9	LC_TIME	name of the ninth month
MON_10	LC_TIME	name of the tenth month
MON_11	LC_TIME	name of the eleventh month
MON_12	LC_TIME	name of the twelfth month
ABMON_1	LC_TIME	abbreviated name of the first month
ABMON_2	LC_TIME	abbreviated name of the second month
ABMON_3	LC_TIME	abbreviated name of the third month
ABMON_4	LC_TIME	abbreviated name of the fourth month
ABMON_5	LC_TIME	abbreviated name of the fifth month
ABMON_6	LC_TIME	abbreviated name of the sixth month
ABMON_7	LC_TIME	abbreviated name of the seventh month
ABMON_8	LC_TIME	abbreviated name of the eighth month
ABMON_9	LC_TIME	abbreviated name of the ninth month

ABMON_10	LC_TIME	abbreviated name of the tenth month
ABMON_11	LC_TIME	abbreviated name of the eleventh month
ABMON_12	LC_TIME	abbreviated name of the twelfth month
RADIXCHAR	LC_NUMERIC	radix character
THOUSEP	LC_NUMERIC	separator for thousands
YESSTR	LC_ALL	affirmative response for yes/no queries
NOSTR	LC_ALL	negative response for yes/no queries
CRNCYSTR	LC_MONETARY	currency symbol, preceded by "-" if the symbol should appear before the value, "+" if the symbol should appear after the value, or "." if the symbol should replace the radix character

File

/usr/include/langinfo.h

Related Information
In this book: "nl_langinfo" in topic 1.2.198.

2.4.11 limits.h

Purpose
Defines implementation specific constants.

Synopsis

#include <limits.h>

Description

This header file defines different categories of names which represent various limits on resources imposed by the system on applications.

For maximum portability, an application should not require more resources than the quantity listed in the "Minimum Acceptable Value" column. An application can have the full amount of a resource available on an implementation by making use of the value given in **limits.h** on that particular system, using the symbolic names listed in the first column of the table.

Because the value of a limit on a particular system may differ from those listed in this header file, an application may use the **pathconf** and **sysconf** functions to determine the actual value of a limit at run-time.

The following constants are defined in **limits.h**:

CHAR_BIT	Number of bits in a char	8
CHAR_MIN	Min integer value of a char	0
CHAR_MAX	Max integer value of a char	+255
UCHAR_MAX	Max value of an unsigned char	+255
SCHAR_MIN	Min value of a signed char	-128
SCHAR_MAX	Max value of a signed char	+127
INT_MIN	Min decimal value of an int	-2147483648
INT_MAX	Max decimal value of an int	+2147483647
UINT_MAX	Max value of an unsigned int	+4294967295
LONG_MIN	Min decimal value of a long	-2147483648
LONG_MAX	Max decimal value of a	+2147483647

	long	
ULONG_MAX	Max value of an unsigned long int	+4294967295
SHRT_MIN	Min decimal value of a short	-32768
SHRT_MAX	Max decimal value of a short	+32767
USHRT_MAX	Max value for an unsigned short	
	int	+65535

pathconf specific constants:

LINK_MAX	Max number of links to a single file	32767
MAX_INPUT	Max number of bytes allowed in a	
	terminal input queue	255
MAX_CANON	Max number of bytes in a terminal	
	canonical input line	255
NAME_MAX	Max number of characters in a filename	
	(not including terminating null)	255
PATH_MAX	Max number of characters in a pathname	
	(not including terminating null)	1023
PIPE_BUF	Max number of bytes that is guaranteed	
	to be atomic when writing to a pipe	40960

sysconf specific constants:

NGROUPS_MAX	Max number of simultaneous supplemen-	
	tary group IDs per process	32

OPEN_MAX	Max number of files that one process	
	can have open at any one time	200
ARG_MAX	Max length of argument to the exec	
	functions including environment data	16384
CHILD_MAX	Max number of processes per user ID	40

POSIX Limit Values:

_POSIX_ARG_MAX	The length of the argument strings for the exec functions	
	in bytes, including environment	
	data	4096
_POSIX_CHILD_MAX	The number of simultaneous	
	processes per real user ID	6
_POSIX_LINK_MAX	The value of a file's link	
	count	8
_POSIX_MAX_CANON	The number of bytes in a	
	terminal canonical input queue	255
_POSIX_MAX_INPUT	The number of bytes for which	
	space will be available in	
	a terminal output queue	255
_POSIX_NAME_MAX	The number of bytes in a	
	filename	14

_POSIX_NGROUPS_MAX	The number of simultaneous supplementary group IDs per process	0
_POSIX_OPEN_MAX	The number of files that one process can have open at one time	16
_POSIX_PATH_MAX	The number of bytes in a pathname	255
_POSIX_PIPE_BUF	The number of bytes that can be written atomically when writing to a pipe	512

Multibyte character specific constant:

MB_LEN_MA	_Max number of bytes in a multibyte character for ANY character code set	4

Defines for message catalog usage:

NL_ARGMAX	Max value of 'digit' in calls to the **printf** and **scanf** subroutines	9
NL_MSGMAX	Max message number	65535
NL_SETMAX	Max set number	65535
NL_TEXTMA	Max number of bytes in a	

	message	
	string	4096

File

/usr/include/limits.h

Related Information
In this book: "pathconf, fpathconf" in topic 1.2.201, and "sysconf" in topic 1.2.296.

2.4.12 locale.h

Purpose
Defines categories.

Synopsis

#include <locale.h>

Description

This header file defines categories as macros which expand to distinct integral-constant expressions to be used as the first argument to the **setlocale** subroutine.

Included in **locale.h** are the categories **LC_ALL**, **LC_COLLATE**, **LC_CTYPE**, **LC_MONETARY**, **LC_NUMERIC**, and **LC_TIME**.

A structure type, **lconv**, used by the **localeconv** subroutine, is also defined in **locale.h**.

Related Information
In this book: "setlocale" in topic 1.2.251, and "localeconv" in topic 1.2.158.

2.4.13 math.h

Purpose
Defines math subroutines and constants.

Synopsis
#include <math.h>

Description
This header file contains declarations of all the subroutines in the Math Library (**libm.a**) and of various subroutines in the Standard C Library (**libc.a**) that return floating-point values.

It defines the structure and constants for the **matherr** error-handling mechanism used by the math subroutines. (See "matherr" in topic 1.2.163 for details about this mechanism.)

Among other things, **math.h** defines the following constant, which is used as an error-return value:

HUGE The maximum value of a single-precision floating-point number.

The following mathematical constants are also defined for your convenience:

M_E The base of natural logarithms (**e**)

M_LOG2E The base-2 logarithm of **e** (log[2] **e**)

M_LOG10E The base-10 logarithm of **e** (log[10] **e**)

M_LN2 The natural logarithm of 2 (log[**e**] 2)

M_LN10 The natural logarithm of 10 (log[**e**] 10)

M_PI &pi., the ratio of the circumference of a circle to its diameter

M_PI_2 The value of &pi. ÷ 2

M_PI_4 The value of &pi. ÷ 4

M_1_PI The value of 1 ÷ &pi.

M_2_PI The value of 2 ÷ &pi.

M_2_SQRTPI The value of 2 divided by the positive square root of &pi.

M_SQRT2 The positive square root of 2

M_SQRT1_2 The positive square root of 1/2.

The **math.h** file contains an **#include** statement that imbeds another header file named **values.h**. This header file defines a number of machine-dependent constants, and it is discussed on page 2.4.28.

Files

/usr/include/math.h
/usr/include/values.h

Related Information
In this book: "matherr" in topic 1.2.163 and "values.h" in topic 2.4.28.

2.4.14 mbcs.h

Purpose
Defines library subroutines used in internationalized programs.

Synopsis

#include <mbcs.h>

Description

The **mbcs.h** header file defines the macro constant **MB_CUR_MAX**, the data type **mbchar_t**, and the macros **_mbdwidth**, **_mblen**, and **_mbsadvance**.

The following are declared as either functions or macros:

mbsadvance	**mbsinvalid**	
mbstomb	**mbscat**	**mbsncat**
mbscpy	**mbsncpy**	**mbschr**
mbsrchr	**mbspbrk**	**mbstok**
mbtowc	**mbstowcs**	**mbslen**
mblen	**mbdwidth**	**mbsspn**
mbscspn	**mbscmp**	**mbsncmp**
wcscat	**wcsncat**	**wcscpy**
wcsncpy	**wcschr**	**wcsrchr**
wcspbrk	**wcstok**	
wcslen	**wcscmp**	**wcsncmp**
wcsspn	**wcscspn**	

Related Information
In this book: "mbstring" in topic 1.2.164, "mbtowc, mbstowcs, mbstomb" in topic 1.2.165, and "wcstring" in topic 1.2.327.

2.4.15 mm

Purpose
Provides the **mm** macro package for formatting documents.

Synopsis

mm [*options*] [*files*]
nroff -mm [*options*] [*files*]
nroff -cm [*options*] [*files*]
mmt [*options*] [*files*]
troff -mm [*options*] [*files*]

Description
This package provides a formatting capability for a very wide variety of documents. How a document is typed and edited on the system is independent of whether the document is to be eventually formatted at a terminal or photoset. See the following references for further details.

Files

/usr/lib/tmac/tmac.m
/usr/lib/tmac/sys.name
/usr/lib/tmac/mm[nt]

Related Information
The **mm**, **mmt**, **nroff**, and **troff** commands in *AIX Operating System Commands Reference*.

2.4.16 mptx

Purpose
Provides the macro package for formatting a permuted index.

Synopsis

nroff -mptx [**flag...**] [**file...**]
troff -mptx [**flag...**] [**file...**]

Description
This package provides a definition for the **.xx** macro which is used for formatting a permuted index produced by the **ptx** program. This package does not provide any other formatting capabilities such as headers and footers. Use this macro package in conjunction with the **mm** macro package for these or other capabilities. In this case, the **-mptx** flag must follow the **-mm** flag. For example:

```
nroff -mm -mptx file
```

or

```
mm -mptx file
```

Files

/usr/lib/tmac/tmac.ptx
/usr/lib/macros/ptx

Related Information
In this book: "mm" in topic 2.4.15.

The **mm, nroff, ptx, troff** commands in *AIX Operating System Commands Reference*.

2.4.17 mv

Purpose
Provides a **troff** macro package for typesetting view graphs and slides.

Synopsis

mvt [**-a**] [**-rw1**] [**flag...**]
[**file...**]
troff [**-a**] [**-rw1**] [**-rX1**] **-mv**
[**flag...**]
[**file...**]

Description
This package makes it easy to typeset view graphs and projection slides in a variety of sizes. A few macros (briefly described in the following) accomplish most of the formatting tasks needed in making transparencies. The facilities of **troff, cw, eqn**, and **tbl** are available for more difficult tasks.

The output can be previewed on most terminals. To preview on some devices, specify the **-rX1** option (this option is automatically specified by the **mvt** command, when that command is invoked with certain options). To preview output on other terminals, specify the **-a** option. The **-rw1** option suppresses the printing of cross-hairs and crop marks.

The available macros are:

.A [**x**] Places text that follows at the first indentation level (left margin); the presence of **x** suppresses the 1/2 line spacing from the preceding text.

.B [**m[s**]]
Places text that follows at the second indentation level. Text is preceded by a mark. **m** is the mark, the default is a large bullet. **s** is the increment or decrement to the point size of the mark with respect to the prevailing point size. The default is 0. If **s** is 100, it causes the point size of the mark to be the same as that of the default mark.

.BX str1 [**str2**] [**f**]
Encloses **str1** in a box and appends **str2** (if any) to it. **str1** is set in the prevailing font unless **f** names a different font.

.C [**m** [**s**]]
Same as **.B**, but for the third indentation level. The default mark is a dash.

.CN [**args**]
Ends a constant-width font display.

.CW [**args**]
Begins a constant-width font display at the current indentation level.

.D [**m** [**s**]]
Same as **.B**, but for the fourth indentation level. The default mark is a small bullet.

.DF n f [**n f...**]

Defines font positions. This may not appear within a foil's input text (for example, it may only appear after all the input text for a foil, but before the next foil-start macro). **n** is the position of font **f**, up to four "**n f**" pairs can be specified. The first font named becomes the prevailing font. The initial setting is (**H** is a synonym for **G**):

```
.DF 1 H 2 I 3 B 4 S
```

.DV [**a**] [**b**] [**c**] [**d**] [**e**]
Alters the vertical spacing between indentation levels. The **a, b, c**, and **d** values alter the spacing for **.A, .B, .C**, and **.D** respectively. The **e** value is the pre-spacing and post-spacing for constant-width font displays bracketed by the **.CW** and **.CN** macros. Arguments that are not null must have dimensions. Null arguments leave the corresponding spacing unaffected. Initial setting is:

```
.DV .5v .5v .5v 0v .5v
```

.I [**in**] [**a** [**x**]]
Changes the current text indent, but does not affect titles. **in** is the indent in inches, unless dimensioned. The default is 0. If **in** is signed, it is an increment or decrement. The presence of **a** invokes the **.A** macro and passes **x**, if any, to it.

.S [**p**] [**l**]
Sets the point size and line length. **p** is the point size, the default is previous. If **p** is 100, the point size reverts to the **initial** default for the current foil-start macro. If **p** is signed, it is an increment or decrement. The default is 18 for **.VS, .VH**, and **.SH**, and 14 for the other foil-start macros. **l** is the line length in inches unless dimensioned. The default is 4.2 inches for **.Vh**, 3.8 inches for **.Sh**, 5 inches for **.SH**, and 6 inches for the other foil-start macros.

.Sh [**n**] [**i**] [**d**]
Same as **.VS**, except that foil size is 5 ¦ 7 inches.

.SH [**n**] [**i**] [**d**]
Same as **.VS**, except that foil size is 7 ¦ 9 inches.

.Sw [**n**] [**i**] [**d**]
Same as **.VS**, except that foil size is 7 ¦ 5 inches.

.SW [**n**] [**i**] [**d**]
Same as **.VS**, except that foils size is 7 ¦ 5.4 inches.

.T string Prints **string** as an over-size, centered title.

.U str1 [**str2**]
Underlines **str1** and concatenates **str2** (if any) to it.

.Vh [**n**] [**i**] [**d**]
Same as **.VS**, except that foil size is 5 ¦ 7 inches.

.VH [**n**] [**i**] [**d**]
Same as **.VS**, except that foils size is 7 ¦ 9 inches.

.VS [**n**] [**i**] [**d**]

Foil-start macro; foil size is to be 7 ¦ 7 inches. **n** is the foil number, **i** is the foil identification, **d** is the date. The foil-start macro resets all parameters (indent, point size, and so on) to initial default values, except for the values of **i** and **d** arguments that came from a previous foil-start macro; it also invokes the **.A** macro.

The naming convention for this and the eight other foil-start macros is that the first character of the name (**V** or **S**) distinguishes between view graphs and slides, respectively, while the second character indicates whether the foil is square (**S**), small wide (**w**), small high (**h**), big wide (**W**), or big high (**H**). Slides are thinner than the corresponding view graphs. For slides, the ratio of the longer dimension to the shorter one is larger than for view graphs. As a result, slide foils can be used for view graphs, but not the opposite. Alternately, view graphs can accommodate a bit more text.

.Vw [**n**] [**i**] [**d**]
Same as **.VS**, except that foil size is 7 inches wide ¦ 5 inches high.

.VW [**n.**] [**i**] [**d**]
Same as **.VS**, except that foil size is 7 ¦ 5.4 inches.

.WS [**w**] [**string**]
Reserves **w** amount of whitespace. **w** must have dimensions. If **string** is present, prints, in the reserved space, the caption:

Paste up string here.

The **.S, .DF, .DV, .U** and **.BX** macros do not cause a break. The **.I** macro causes a break only if it is invoked with more than one argument. All the other macros cause a break.

The macro package also recognizes the following uppercase synonyms for the corresponding lower case **troff** requests:

.AD .BR .CE .HY .NA .NH .NX .SO .SP .TA .TI

The **Tm** string produces the trademark symbol.

The ~ (tilde) character is translated into a blank on output.

The following **troff** symbols are defined:

***t** The ASCII tab character.

***E** The ellipsis (...). Do not use this symbol within constant-width text.

***u** The short name of the operating system in small capital letters.

***(UU** The short name of the operating system with a leading full-cap letter.

***(UF** The full name of the operating system.

***(Tm** The trademark symbol.

Note: The VW and SW foils are meant to be 9 inches wide by 7 inches high. However, the typesetter paper is generally only 8 inches wide, so they are printed 7 inches wide by 5.4 inches high. They need to be enlarged by a factor of 9/7 before they can be used as view graphs.

Files

/usr/lib/tmac/tmac.v
/usr/lib/macros/vmca

Related Information
The **cw**, **eqn**, **mmt**, **tbl**, and **troff** commands in *AIX Operating System Commands Reference*.

2.4.18 netgroup

Purpose
Provides list of network groups.

Description
The **netgroup** facility defines network wide groups, used for permission checking when doing remote mounts, remote logins and remote shells. For remote mounts, the information in **netgroup** is used to classify machines; for remote logins and remote shells, it is used to classify users. Each line of the netgroup file defines a group and has the following format:

```
groupname member1 member2...
```

where **member1** is either another group name or:

```
(hostname, username, domainname)
```

Any of these three fields may be empty, in which case it signifies a wildcard. For example:

```
biggroup (, ,)
```

defines a group to which everyone belongs.

A gateway machine should be listed under all possible names by which it may be recognized.

```
ying (gateway, ,) (gateway-rlt, ,)
```

Field names that begin with something other than a letter, digit or underscore (such as a hyphen) work in the opposite fashion. For example, consider the following entries:

```
groupa (sitea,-,la)
groupb (-,george,la)
```

The machine **sitea** belongs to the group **groupa** in the domain **la**, but no users belong to it. Similarly, the user **george** belongs to the group **groupb** in the domain **la**, but no machines belong to it.

The **domainname** field refers to the domain **n** where the entry is valid, not the name containing the trusted host.

Files
/etc/netgroup

Related Information
The **makedbm** and **ypserv** commands in *AIX Operating System Commands Reference*.

2.4.19 nl_types.h

Purpose
Contains definitions of data types.

Synopsis

#include <nl_types.h>

Description

This header file contains definitions for the following file types:

nl_catd Used by the message catalog functions **catopen**, **catgets**, and **catclose** to identify a catalog descriptor.

nl_item Used by **nl_langinfo** to identify items of **langinfo** data. Values of objects of type **nl_item** are defined in **langinfo.h**.

Related Information
In this book: "langinfo.h" in topic 2.4.10

The **catclose**, **catgets**, and **catopen** subroutines in *AIX Operating System Technical Reference*.

2.4.20 param.h

Purpose
Describes system parameters.

Synopsis
#include <sys/param.h>

Description
Parameters vary among systems using the AIX Operating System. For AIX/370 and AIX PS/2, these parameters are in the **/sys/param.h** file. The most significant parameters are:

BSIZE Indicates the kernel buffer size. Both AIX/370 and AIX PS/2 have a buffer size of 4096 bytes.

NOFILE Indicates the maximum number of open files allowed per process. This value is 200.

NCARGS Indicates the maximum number of characters, including terminating NULLs that may be passed using the **exec** system call.

File
/usr/include/sys/param.h

2.4.21 stdarg.h

Purpose
Defines variable argument list access macros.

Synopsis

```
#include <stdarg.h>
```

Description

The **stdarg.h** header file defines the data type **va_list** and the following macros used for advancing through a list of arguments:

```
#include <stdarg.h>

void va_start (va_list ap, parmN);
type va_arg (va_list ap, type);
void va_end (va_list ap);
```

Related Information
In this book: "varargs" in topic 1.2.323.

2.4.22 stat.h

Purpose
Defines the data structure returned by the **statx, fstatx, stat, fstat, fullstat, ffullstat,** and **lstat** system calls.

Synopsis

```
#include <sys/stat.h>
```

Description
The **statx** and **fstatx** system calls obtain information about a file. These system calls return a data structure defined by the **<sys/stat.h>** include file.

```
  struct stat
  {
  /* Beginning of stat structure replica ... */
      dev_t       st_dev;         /* Global file system (gfs)     */
                                  /* number of the file system    */
                                  /* containing a directory       */
                                  /* entry for this file. File    */
                                  /* index+gfs number uniquely    */
                                  /* identifies the file within   */
                                  /* the system.                  */
      ino_t       st_ino;         /* File index number (inode no.)*/
      mode_t      st_mode;        /* File mode; see below         */
      nlink_t     st_nlink;       /* Number of links              */
      u_short_t   st_spare0;      /* Reserved                     */
      uid_t       st_uid;         /* User ID of the file's owner  */
      gid_t       st_gid;         /* Group ID of the file's group */
      dev_t       st_rdev;        /* ID of device                 */
                                  /* This entry is defined only   */
                                  /* for character or block       */
                                  /* special files.               */
      off_t       st_size;        /* File size in bytes           */


      /* Times are seconds since 00:00:00 GMT, Jan. 1, 1970 */
      time_t      st_atime;       /* Time of last access          */
      u_long_t    st_spare1;
      time_t      st_mtime;       /* Time data last modified      */
      u_long_t    st_spare2;
      time_t      st_ctime;       /* Time file status last changed */
      u_long_t    st_spare3;

      u_long_t    st_blksize;     /* Size of block in file        */
      u_long_t    st_blocks;      /* Number of blocks in file     */

      u_long_t    st_gen;         /* Inode generation number      */
      u_long_t    st_type;        /* Vnode type                   */

  #        define  VNON       0
  #        define  VBAD       1
  #        define  VREG       2
  #        define  VDIR       3
  #        define  VHDIR      4
  #        define  VBLK       5
  #        define  VCHR       6
  #        define  VLNK       7
```

```
#       define  VSOCK      8
#       define  VFIFO      9
#       define  VMPC      10

    u_long_t   st_vfs;          /* vfs id                        */
    u_long_t   st_flag;         /* flag word                     */
#       define  FS_NOFLAG 0x00 /* clear flag                    */
#       define  FS_MOUNT  0x01 /* this file is the root of a    */
                               /* file system or is mounted     */
                               /* over                          */
#       define  FS_REMOTE 0x02 /* file is remote                */
#       define  FS_VMP    FS_MOUNT

    /* The following fields are used by TCF */
    u_long_t   st_cmtcnt;       /* gfs commit sequence number    */
    fstore_t   st_fstore;       /* file-replication storage      */
                               /* attribute                     */
    long       st_version;      /* version number of file        */
    siteno_t   st_css;          /* current synchronization site */
    siteno_t   st_ss;           /* current storage site (fstat) */
    siteno_t   st_rdevsite;     /* rdev site (devices only)      */
    short      st_spare4;


/* ... end of old stat structure replica ... */
#   define STATSIZE \
    (((int)(char*)&(((struct stat *)0)->st_spare4))+(sizeof(short)))

/*
 * the stat() compatibility interface does not
 * return any of the following fields.
 */

/* ... beginning of fullstat structure, used by DS ... */

    long       st_nid;          /* Node id                       */
    uid_t      st_uid_raw;      /* The file's untranslated uid   */
    gid_t      st_gid_raw;      /* The file's untranslated gid   */
    u_long_t   st_uid_rev_tag;  /* uid translation tag           */
    u_long_t   st_gid_rev_tag;  /* gid translation tag           */

#define IDTAG_CALLER     1      /* definitions for _rev_tag      */
#define IDTAG_OTHER      2      /* fields                        */
#define IDTAG_SOMEONE    3
#define IDTAG_NO_ONE     4

/* ... End of fullstat structure replica */
#   define FULLSTATSIZE \
    (((int)(char*)&(((struct stat *)0)->st_gid_rev_tag))+sizeof(u_long_t))

}

/*
 * Defines for statx cmd argument
 */
#define STX_LINK        0x0001  /* do not traverse final symbolic */
                                /* link (lstat)                   */
#define STX_MOUNT       0x0002  /* If a mount point return status */
                                /* of mounted-over directory      */
#define STX_HIDDEN      0x0004  /* do not traverse final hidden   */
```

```
                                /* directory                        */

#define STX_TRANS       0x0000  /* Normal uid/gid translation     */
#define STX_TRANS_NONE  0x0008  /* No uid/gid translation         */
#define STX_TRANS_OTHER 0x0010  /* Biased uid/gid translation     */


/*
 * Defines for file types in st_mode
 */
#define S_IFMT   0x3000F000       /* type of file */
#define S_IFDIR  0x00004000       /* directory */
#define S_ISDIR(m)  (((m) & (S_IFMT)) == (S_IFDIR))
#define S_IFCHR  0x00002000       /* character special */
#define S_ISCHR(m)  (((m) & (S_IFMT)) == (S_IFCHR))
#define S_IFBLK  0x00006000       /* block special  */
#define S_ISBLK(m)  (((m) & (S_IFMT)) == (S_IFBLK))
#define S_IFREG  0x00008000       /* regular */
#define S_ISREG(m)  (((m) & (S_IFMT)) == (S_IFREG))
#define S_IFIFO  0x00001000       /* fifo (named pipe) */
#define S_ISFIFO(m)  (((m) & (S_IFMT)) == (S_IFIFO))
#define S_IFLNK  0x0000a000       /* symbolic link (lstat only) */
#define S_ISLNK(m)  (((m) & (S_IFMT)) == (S_IFLNK))
#define S_IFSOCK 0x0000c000       /* socket */
#define S_ISOCK(m)  (((m) & (S_IFMT)) == (S_IFSOCK))


/*
 * file attributes in st_mode
 */
#define S_ISUID  0x00000800       /* set user id on execution     */
#define S_ISGID  0x00000400       /* set group id on execution    */
#define S_IMPX   0x00000200       /* multiplexed device           */
#define S_ISVTX  0x00000200       /* save swapped text even after */
                                  /* use                          */
#define S_IHIDDEN 0x08000000      /* hidden directory             */
#define S_ENFMT  S_ISGID          /* record locking enforcement   */
                                  /* flag                         */

#define S_IFMPX  (S_IFCHR | S_IMPX) /* mode for multiplexed file  */

#define S_ISMPX(m) (((m) & (S_IFMT | S_IMPX)) == (S_IFMPX))
#define S_ISHIDDEN(m) (((m) & (S_IFMT | S_IHIDDEN)) == \
                                          (S_IFDIR | S_IHIDDEN))


/*
 * file permissions in st_mode
 */
#define S_IRWXU  00700  /* (0x01C0) owner read,write,execute */
                                  /*  permission              */
#define S_IREAD  00400  /* (0x0100) owner read permission    */
#define S_IRUSR  00400  /* (0x0100) read permission, owner   */
#define S_IWRITE 00200  /* (0x0080) owner write permission   */
#define S_IWUSR  00200  /* (0x0080) owner write permission   */
#define S_IEXEC  00100  /* (0x0040) owner execute/search permission */
#define S_IXUSR  00100  /* (0x0040) owner execute/search permission */

#define S_IRWXG  00070  /* (0x0038) group read,write,execute */
                        /* permission                        */
```

```
#define S_IRGRP  00040  /* (0x0020) group read permission    */
#define S_IWGRP  00020  /* (0x0010) group write permission   */
#define S_IXGRP  00010  /* (0x0008) group execute/search permission */

#define S_IRWXO  00007  /* (0x0007) other read,write,execute */
                        /* permission                        */
#define S_IROTH  00004  /* (0x0004) other read permission    */
#define S_IWOTH  00002  /* (0x0002) other write permission   */
#define S_IXOTH  00001  /* (0x0001) other execute/search permission */
```

The meanings of the **stat** structure fields are:

st_dev The gfs number of the file system containing a directory entry for this file. The file index together with the gfs number uniquely identifies the file within the system.

st_ino The index (inode number) of this file in this file system. A file is uniquely identified by specifying the file system on which it resides and its inode number in this file system.

st_mode The file mode. The values of this field are described above.

st_nlink The number of hard links to the file. See "link" in topic 1.2.156.

st_size The end-of-file mark. For a regular file or directory, **st_size** specifies the length of the file in bytes; for a FIFO or pipe, it specifies the number of unread bytes. For a device, **st_size** is undefined.

st_rdev The ID of the device. This field is defined only for block or character special files.

st_atime The time when file data was last accessed.

st_mtime The time when data was last modified.

st_ctime The time when file status was last changed.

st_blksize
The size, in bytes, of each block of the file.

st_blocks The number of blocks used to represent the file on permanent storage. This includes indirect blocks.

st_gen The generation number of this inode.

st_type The type of the vnode for this object. This is one of the following values:

VNON An unallocated object; this should not occur

VBAD An unknown type of object

VREG regular file

VDIR directory

VHDIR hidden directory

VBLK block device

VCHR character device

VLNK symbolic link

VSOCK socket

VFIFO FIFO

VMPC multiplexed character device

st_vfs Virtual file system ID.

st_flag A flag indicating whether the file or directory is a mount point. A value of FS_MOUNT indicates that it is a mount point.

st_uid The file owner ID.

st_gid The file group ID.

File
/usr/include/sys/stat.h

Related Information
In this book: "statx, fstatx, stat, fstat, fullstat, ffullstat, lstat" in topic 1.2.282 and "types.h" in topic 2.4.27.

2.4.23 stddef.h

Purpose
Lists common definitions.

Synopsis

```
#include <stddef.h>
```

Description

The **stddef.h** header file defines the macro constant **NULL**, the types **ptrdiff_t**, **size_t**, and **wchar_t**, and the macro **offsetof** (**type, member-designator**).

2.4.24 stdlib.h

Purpose
Lists standard library definitions.

Synopsis

```
#include <stdlib.h>
```

Description

The **stdlib.h** header file defines the macro constants **RAND_MAX**, **EXIT_SUCCESS**, **EXIT_FAILURE**, **NULL**, and **MB_CUR_MAX** and the data types **size_t**, **wchar_t** and **mbchar_t**, defined through **typedef**.

The following are declared as either functions or macros:

abort	**abs**	**atof**	**atoi**
atol	**bsearch**	**calloc**	**exit**
free	**getenv**	**labs**	**malloc**
mblen	**mbtowc**	**mbstowcs**	**srand**
qsort	**rand**	**realloc**	
strtod	**strtol**	**system**	
wcstombs	**wctomb**		

2.4.25 string.h

Purpose
Defines string operations.

Synopsis

```
#include <string.h>
```

Description

The **string.h** header file defines the macro constant **NULL** and the data type **size_t** defined through **typedef**.

The following are declared as either subroutines or macros:

strcpyn	**strcatn**	**strcmpn**	
strcpy	**strncpy**	**strcat**	
strncat	**strchr**	**strrchr**	
strstr	**strcmp**	**strncmp**	
strcspn	**strlen**	**strspn**	
memchr	**memcpy**	**memset**	**memcmp**
index	**rindex**		
strpbrk	**strtok**	**strtol**	
NLstrcpy	**NLstrncpy**	**NLstrcat**	**NLstrncat**
NLstrchr	**NLstrrchr**	**NLstrpbrk**	**NLstrtok**
***NCstrcpy**	***NCstrncpy**	***NCstrcat**	***NCstrncat**
***NCstrchr**	***NCstrrchr**	***NCstrpbrk**	***NCstrtok**

Related Information
In this book: "NLstring" in topic 1.2.193 and "string" in topic 1.2.288.

2.4.26 TERM

Purpose
Lists conventional names for terminals.

Description
These names are used primarily for commands such as **mm** and **nroff**. These names are maintained as part of the shell environment in the variable **TERM**. See the **sh** command in *AIX Operating System Commands Reference* for an explanation of the shell. Also see "profile" in topic 2.3.48 and "environment" in topic 2.4.6 in this book for use of the **TERM** environment variable.

TERM	**Terminal Description**
ibm3161	IBM 3161 ASCII Display Station
ibm3161	IBM 3163 ASCII Display
ibm3161-C	IBM 3161 ASCII Display Station with cartridge (for international character support)
ibm3162	IBM 3162 ASCII Display (for international character support)
ibm8503	IBM 8503 Display
ibm8507	IBM 8507 Display
ibm8512	IBM 8512 Display
ibm8513	IBM 8513 Display
ibm8514	IBM 8514 Display
ibm8604	IBM 8604 Display
vt100	DEC VT100
vt220	DEC VT220
dumb	Terminal types with no special features (such as reverse line motion) (implies **-c**)
lp	Line Printer (implies **-c**) (must pipe through **lpr** or some such filter)
37	Teletype Model 37 KSR
42	ADM 42 (implies **-c**)
300	DASI (DTC, GSI) 300
300s	DASI 300s
300-12	DASI 300 at 12-pitch
300s-12	DASI 300s at 12-pitch
tn300	TermiNet 300 (implies **-c**)
382	DTC 382
450	DASI 450 (same as Diablo 1620) DEFAULT
450-12	DASI 450 (same as Diablo 1620) 12-pitch
2631	HP 2631 series line printer (implies **-c**)
2631-e	HP 2631 series (expanded mode) (implies **-c**)
2631-c	HP 2631 series (compressed mode) (implies **-c**)
4000a	Trendata 4000a

Up to eight characters chosen from **[-a-z0-9]** make up a basic terminal name. Terminal sub-models and operational modes are distinguished by suffixes beginning with a **-** (hyphen). Names should generally be based on original vendors, rather than local distributors. A terminal acquired from one vendor should not have more than one distinct basic name.

Commands whose behavior depends on the type of terminal should accept parameters such as **-Tterm** where **term** is one of the names in the preceding list. If the parameter is not in the list, the commands should obtain the terminal type from the environment variable TERM, which in turn should contain **term**. Any unknown terminal is treated as a **dumb** terminal.

This list does not include all supported terminals. See "terminfo" in

topic 2.3.59 for additional TERM variables.

File
usr/lib/help/term

Related Information
In this book: "environment" in topic 2.4.6 and "terminfo" in topic 2.3.59.

The **mm, sh, stty, tabs, nroff**, and **environ** commands in *AIX Operating System Commands Reference*.

2.4.27 types.h

Purpose
Defines data types for the system.

Synopsis
#include <sys/types.h>

Description
The data types defined in this include file are used in the AIX system source code. Some data of these types are accessible to user code:

```
typedef struct {int r[1];} * physadr;
typedef long                 daddr_t;
typedef char *               caddr_t;
typedef unsigned int         uint;
typedef unsigned short       ushort;
typedef unsigned long        ulong;
typedef ulong                ino_t;
typedef long                 time_t;
typedef ulong                dev_t;
typedef long                 off_t;
typedef long                 paddr_t;
typedef long                 key_t;
typedef long                 pid_t;
typedef ulong                uid_t;
typedef ulong                gid_t;
typedef ulong                mode_t;
```

Notes:

daddr_t This data type is used for disk addresses.

time_t Times are encoded in seconds since 00:00:00 GMT, January 1, 1970.

dev_t The major and minor parts of a device code specify kind of device and unit number of the device, and they depend on the system customization.

off_t Offsets are measured in bytes from the beginning of a file.

pid_t Process ID type.

uid_t User ID type.

gid_t Group ID type.

mode_t File mode and permission bits.

File

/usr/include/sys/types.h

Related Information
In this book: "fs" in topic 2.3.20 and "values.h" in topic 2.4.28.

2.4.28 values.h

Purpose
Defines machine-dependent values.

Synopsis
#include <values.h>

Description
This header file contains a set of manifest constants that are conditionally defined for particular processor architectures. The model for integers is assumed to be a ones- or twos-complement binary representation, in which the sign is represented by the value of the high-order bit.

BITS(type) The number of bits in the specified data type

HIBITS A short integer with only the high-order bit set (0x8000)

HIBITL A long integer with only the high-order bit set (0x80000000)

HIBITI A regular integer with only the high-order bit set (the same as **HIBITL**)

MAXSHORT The maximum value of a signed short integer (0x7FFF == 32767)

MAXLONG The maximum value of a signed long integer (0x7FFFFFFF == 2147483647)

MAXINT The maximum value of a signed regular integer (the same as **MAXLONG**)

MAXFLOAT The maximum value of a single-precision floating-point number

MAXDOUBLE The maximum value of a double-precision floating-point number

LN_MAXDOUBLE The natural logarithm of **MAXDOUBLE**

MINFLOAT The minimum positive value of a single-precision floating-point number

MINDOUBLE The minimum positive value of a double-precision floating-point number

FSIGNIF The number of significant bits in the mantissa of a single-precision floating-point number

DSIGNIF The number of significant bits in the mantissa of a double-precision floating-point number

FMAXEXP The maximum exponent of a single-precision floating-point number

DMAXEXP The maximum exponent of a double-precision floating-point number

FMINEXP The minimum exponent of a single-precision floating-point

number

DMINEXP The minimum exponent of a double-precision floating-point number

FMAXPOWTWO The largest power of two that can be exactly represented as a single-precision floating-point number

DMAXPOWTWO The largest power of two that can be exactly represented as a double-precision floating-point number.

File

/usr/include/values.h

Related Information
In this book: "math.h" in topic 2.4.13 and "types.h" in topic 2.4.27.

2.5 Chapter 5. Special Files

Subtopics

2.5.1 About This Chapter

This chapter describes various special files that refer to specific hardware peripherals and AIX system device drivers. The names of the entries are generally derived from names for the hardware as opposed to the names of the special files themselves. Characteristics of both the hardware device and the corresponding AIX system device driver are discussed where applicable.

The special files in this chapter are categorized in the following manner:

Special files found in all AIX systems. These are:

error
mem, kmem
null
osm
prf
pty
termio
trace
tty

Special files unique to AIX/370. These are:

ceti
ckd
cpcmd
fba
ilans
lp
mt
punch
reader

Special files unique to AIX PS/2. These are:

asy
fd
hd
hft
keyboard
lp
nvram
tape

2.5.2 asy

Purpose
Supports asynchronous serial ports.

Description
The **asy** driver supports asynchronous serial ports. It is unique to AIX PS/2. If a port is not installed, an attempt to open it fails. Each port can be individually programmed for speed (50-19.2K baud), character length, and parity. Output speed is always the same as input speed. This driver supports the PS/2 system board serial port, the IBM PS/2 Dual Async Adapter/A, and the IBM PS/2 300/1200 Internal Modem/A.

The asynchronous port is a character-at-a-time device for both input and output. This characteristic limits the bandwidth, which can be achieved over a line and increases the interrupt loading on the central processor.

If the port was opened with the modem control bit present in the minor device (see the following text), modem control is enabled. If enabled, the driver waits in the **open** routine until data carrier detect is present. Once opened, if data carrier detect drops, the driver returns errors on any subsequent user read or write attempts of the asynchronous port. If the port was opened as a controlling teletype, a **SIGHUP** signal is generated to the process that performed the open.

If the port was opened with the printer control bit present in the minor device number (see the following text), the port behaves as a serial printer port and the driver interface is described by **lp**. If the printer control bit is not present, the port behaves as a terminal port and the driver interface is described by **termio**.

Subtopics
2.5.2.1 Minor Device Numbers

2.5.2.1 Minor Device Numbers

The values of the low-order bits of the minor device number correspond to the port addresses of the serial ports as configured by the IBM PS/2 Reference Diskette. These values are in the range of 0 through 7 and indicate serial ports SERIAL_1 through SERIAL_8, respectively. Bit 6 enables modem control on the selected port. Bit 7 indicates that the selected port is a serial printer port. Following are some examples of minor device numbers.

Minor device	**Meaning**
0	port SERIAL_1, no modem control, **tty**
7	port SERIAL_8, no modem control, **tty**
64	port SERIAL_1, modem control, **tty**
65	port SERIAL_2, modem control, **tty**
129	port SERIAL_2, no modem control, **lp**
193	port SERIAL_2, modem control, **tty**

Files

/dev/tty* for terminal devices.

/dev/lp* for printer devices.

Related Information
In this book: "sigaction, sigvec, signal" in topic 1.2.263, "lp" in topic 2.5.14, and "termio" in topic 2.5.28.

The **devices** command in *AIX Operating System Commands Reference*.

2.5.3 cdrom

Purpose

Supports the **cdrom** device driver.

Description

The **cdrom** device driver provides block and character access to the **cdrom** disks. The **cdrom** special file is unique to AIX PS/2. The minor number of this system device is determined via the special files **/dev/cdromn** and **/dev/rcdromn**, where *n* specifies the minor number.

In raw I/O, the buffer must begin on a 512 byte boundary and counts must be a multiple of 512 bytes. Likewise **lseek** system calls must specify a multiple of 512 bytes. However, for the most efficient raw I/O, the buffer should be on a 4096 byte boundary and counts should be a multiple of 4096 (the page size). Note that **cdrom** devices are read only; thus, any writes will return an error.

Subtopics
2.5.3.1 ioctl Operations

2.5.3.1 ioctl Operations

The **IOCTYPE** type **ioctl** call returns the value *DD_CDROM,* defined in **/sys/devinfo.**

The **IOCTYPE** type **ioctl** call returns the structure defined in **/sys/devinfo.h.**

Files

/dev/cdrom0, /dev/cdrom1, ...
/dev/rcdrom0, /dev/rcdrom1, ...

Related Information
In this book: "fd" in topic 2.5.9 and "ioctlx, ioctl, gtty, stty" in topic 1.2.137.

2.5.4 ceti

Purpose
Supports the CETI network device driver.

Synopsis

```
#include <sys/devinfo.h>
#include <sys/b370/ceti.h>
```

Description
The **ceti** device driver is used to drive to the 9370 Integrated Ethernet Adapter, the Intel Fastpath Ethernet Adapter, and other ethernet adapters that conform to the CETI interface specification. This driver is unique to AIX/370.

Since the supported device is a LAN network interface, normal network traffic is transmitted and received using the socket interface and not the device interface. The device interface is used only to monitor and control the operation of the device, via **ioctl** commands.

The device used to perform this monitor and control is **/dev/ceti#**, where # is the device minor number.

There is a small number of IOCTL operations available besides the standard IOCTYPE and IOCINFO commands. They are:

DIOCONLINE
Bring the device on-line.

```
ioctl(fd, DIOCONLINE, arg)
char *arg
```

The argument is ignored.

DIOCOFFLINE
Take the device off-line.

```
ioctl(fd, DIOCOFFLINE, arg)
char *arg
```

The argument is ignored.

CIOGETSTATS
Return device statistics.

```
ioctl(fd, CIOGETSTATS, arg)
struct ceti_lstat *arg;
```

where **struct ceti_lstat** is declared as:

```
struct ceti_lstat {
    ulong received;          /* # of frames received */
    ulong transmitted;       /* # of frames transmitted */
    ulong lost;              /* # of frames lost */
    ulong runts;             /* # of runt frames */
    ulong crcerrs;           /* # of frames with crc errors */
    ulong collisions;        /* # of frames in collision */
    ulong multicast;         /* # of multicast frames */
    ulong broadcast;         /* # of broadcast frames */
```

```
    ulong not_ours;          /* # of frames not for us */
    ulong multi_not_ours;    /* # of multicast frames not for us *
    ulong excess_collisions; /* # of excessive collisions */
    ulong out_of_window;     /* # of out-of-window collisions */
    ulong alignment;         /* # of alignment errors */
    ulong shorted;           /* # of times a short was detected */
    ulong reflectometer;     /* # of reflectometer packets */
    ulong too_long;          /* # of frames that were too long */
};
```

CEREPCTL

Control the VM EREP logging.

```
ioctl(fd, CEREPCTL, arg)
struct ceti_erep_ctl *arg;
```

where **struct ceti_erep_ctl** is declared as:

```
struct ceti_erep_ctl {
    unsigned char obr_thresh;
    unsigned char mdr_thresh;
    unsigned char obr_cnt;
      /* current log control count returned here */
    unsigned char mdr_cnt;
      /* current log control count returned here */
};
```

If a threshold is 0, no EREP logging is done. If set to 0xff, no change to the current state is made. Otherwise, logging set to occur every **n**th occurrence, where **n** is the number specified.

Related Information

In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137 and "open, openx, creat" in topic 1.2.199.

2.5.5 ckd

Purpose
Supports the Count Key Data Direct Access Storage Device driver.

Synopsis

```
# include <sys/devinfo.h>
# include <sys/b370/dkckd.h>
```

Description
Fixed disk devices on an AIX/370 system provide block and character access to minidisks on physical disk devices. The **ckd** special file is unique to AIX/370.

The particular device accessed is of the form **/dev/chdnnn** and **/dev/rchdnnn**, where **nnn** is the given device minor number. The driver supports up to 32 separate AIX minidisks (partitions) on a single physical disk. Thus, each physical disk has 32 minor numbers associated with it. The first of these refers to the entire disk and is normally used for disk maintenance only. The other 31 minors are available for user or system AIX minidisks.

In raw I/O, the buffer must always be aligned on a 4096 byte boundary, and counts must be a multiple of 4096 bytes (an integral number of physical blocks). **lseek** system calls should also specify such an aligned address.

A number of IOCTL operations are available. In addition to IOCTYPE and IOCINFO, the following calls are defined:

IOCSTATS
Returns device statistics.

```
ioctl(fd, IOCSTATS, arg);
struct fi_status *arg;
```

where **fi_stats** is declared as:

```
struct fi_stats {
    u_long Fi_nsio;     /* number read + write sio's  */
    u_long Fi_nblksrd;  /* number blocks read         */
    u_long Fi_nblkswr;  /* number blocks written      */
    u_long Fi_nretry;   /* number of retries          */

    u_long Fi_tsio;     /* rmtime for last sio        */
    double Fi_etsio;    /* elapsed time for sio       */
    double Fi_tottsio;  /* microsecs from sio to intr */

    u_long Fi_nfree;    /* number of ckdfree calls    */
    u_long Fi_tfree;    /* rmtime for last ckdfree    */
    double Fi_etfree;   /* elapsed time for ckdfree   */

    u_long Fi_nrsort;   /* number rotate_sort calls   */
    u_long Fi_trsort;   /* rmtime last rotate_sort    */
    double Fi_etrsort;  /* rotate_sort elapsed time   */
};
```

DEVADDR
Returns the device address.

```
ioctl(fd, DEVADDR, arg);
int *arg;
```

HDIOPAR

Returns driver internal structure.

```
ioctl(fd, HDIOPAR, arg);
struct ckd_info *arg;
```

where **struct ckd_info** is declared as:

```
 struct ckd_info {
    struct buf   fi_tab;
      /* Queue header, should be a bufhdr? */
    ioaddr_t     fi_devno;
      /* Device number for this spindle */
    u_char       fi_mask;
       /* Mask byte for set file mask ccw */
    char         fi_unused;
    bool_t       fi_alive;
       /* Is it really there ? */
    bool_t       fi_ro;
       /* Disk is read-only (How do I know?) */
    bool_t       fi_countkeyio;
       /* Open for count key I/O */
    short        fi_ocount;
       /* Open count */
    short        fi_rocount;
       /* Raw open count */
    short        fi_recptrk;
       /* Records per track  */
    short        fi_trkpcyl;
       /* Tracks per cylinder */
    daddr_t      fi_curblock;
       /* Current head position */
    daddr_t      fi_maxsecno;
       /* Maximum number of sectors */
    short        *fi_sectabp;
       /* Ptr to record->sector table*/
    ccw_t        *fi_realccw;
       /* Real address of the ccw's */
    ccw_t        *fi_ccws;
       /* Ptr to channel program */
    seeklist_t   *fi_seek;
       /* Ptr to seek or search addresses */
    paddr_t      **fi_idaws;
       /* Ptr to space for idaws */
    ccw_t        *fi_insccwp;
       /* Ptr to sense channel program */
    u_char       *fi_snsp;
       /* Ptr to buffer for sense data */
    int          fi_nsns;
       /* Number of sense bytes retrieved */
    struct fi_stats     fi_stats;
       /* Statistics can be used by sar */
          };
```

HDIORST

Reread the VTOC minidisk partition table.

```
ioctl(fd, HDIORST, arg);
caddr_t arg;
```

The argument is not used. This command only works on the whole partition maintenance minidisk.

Related Information

In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "open, openx, creat" in topic 1.2.199, "read, readv, readx" in topic 1.2.224, and "write, writex" in topic 1.2.330.

2.5.6 cpcmd

Purpose
Supports the IBM VM CP system interface.

```
# include <sys/devinfo.h>
# include <sys/b370/cpcmd.h>
```

Description
The AIX/370 system runs as a guest under VM. At times, it is desirable for programs to communicate through this interface. The device used to obtain this interface to VM is **/dev/cpcmd**. The **cpcmd** special file is unique to AIX/370. There is only one such device per system.

Only the superuser can open this device. This avoids giving non-privileged AIX users the full privileges of the AIX/370 virtual machine within the realm of VM. Most I/O through this device is performed via **ioctl** commands describe below. It is possible, however, to write to the device a specific CP command. This command should be in ASCII, but otherwise in the format required by VM (for instance, all uppercase letters).

There is one **IOCTL** operation available besides the standard **IOCTYPE** and **IOCINFO** commands. This is the **CPCMD ioctl** command. The **CPCMD ioctl** is of the form:

```
ioctl (fd, CPCMD, arg)
struct cpcmds *arg;
```

The **cpcmd** structure is declared as follows:

```
struct cpcmds {
  int   inlen;
  int   maxoutlen;
  char  *inbuf;
  char  *outbuf;
  int   *outlen;
};
```

The **inbuf** contains the command to be run, and the **outbuf** is to store the result returned by CP. This method allows the execution of each CP command to be atomic.

Error Conditions
In addition to the errors listed in "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "open, openx, creat" in topic 1.2.199, and "write, writex" in topic 1.2.330, system calls to this device can fail in the following circumstances:

EPERM Attempt to open the device by non-superuser.

EINVAL The CP command is not valid.

Related Information
In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "open, openx, creat" in topic 1.2.199, and "write, writex" in topic 1.2.330.

The **cpcmd** command in *AIX Operating System Commands Reference*.

2.5.7 error

Purpose
Logs system events.

Synopsis

#include <sys/erec.h>

Description
The **error** special file is found in all AIX systems.

The format of an event record depends on the type of event encountered. Each record, however, has a header with the following format:

```
  struct errhdr {
     int       e_len;       /* word in record (with header) */
     time_t    e_time;      /* time of day */
     long      e_timex;     /* clock ticks */
     char      e_nid[8];    /*same as nodename field returned by uname sys cal
     char      e_vmid[8];   /*same as sysname field returned by uname sys call
      union {
         struct  {
            char  ex_class;
            char  ex_subclass[2];
            char  ex_type;  /* record type */
         } ex;
         int csmt;
      } exx;
  };
```

The error daemon searches the RAS configuration file **/etc/rasconf** for a stanza labeled **/dev/error**. Minor device 0 of the **error** driver is the interface between a process and the routines that collect error-records in the system. This driver can be opened only for reading by a process (usually the error daemon) with superuser permission. Each read retrieves an entire error record. A read request of less than the entire record causes the retrieved record to be truncated. Multiple processes can open the **error** file to write.

File
/dev/error

Related Information
In this book: "errunix" in topic 1.2.70, and "rasconf" in topic 2.3.50.

The **errdemon** command in *AIX Operating System Commands Reference*.

2.5.8 fba

Purpose
Supports the Fixed Block Architecture Direct Access Storage Device driver.

Synopsis

```
# include <sys/devinfo.h>
# include <sys/b370/dkfba.h>
```

Description
Fixed disk devices on an AIX/370 system provide block and character access to minidisks on physical disk devices. The **fba** special file is unique to AIX/370.

The particular device accessed is of the form **/dev/fhdnnn** and **/dev/rfhdnnn**, where **nnn** is the given device minor number. The driver supports up to 32 separate AIX minidisks (partitions) on a single physical disk. Thus, each physical disk has 32 minor numbers associated with it. The first of these refers to the entire disk and is normally used for disk maintenance only. The other 31 minors are available for user or system AIX minidisks.

In raw I/O, the buffer must always be aligned on a 4096 byte boundary, and counts must be a multiple of 4096 bytes (an integral number of physical blocks). **lseek** system calls should also specify such an aligned address.

A number of IOCTL operations are available. In addition to IOCTYPE and IOCINFO, the following calls are defined:

DEVADDR Returns the device address

```
ioctl(fd, DEVADDR, arg);
int *arg;
```

HDIOPAR Return driver internal structure

```
ioctl(fd, DEVADDR, arg);
struct fba_info *arg;
```

where **struct fba_info** is declared as:

```
struct fba_info {
  struct buf   fi_tab;
         /* Queue header */
  bool_t       fi_alive;
         /* Is it really there ? */
  bool_t       fi_ro;
         /* Disk is read-only */
  short        fi_ocount;
         /* Open count */
  short        fi_rocount;
         /* Raw open count */
  ioaddr_t     fi_daddr;
         /* I/O address for this device */

  int          fi_physec;
        /* Physical sector size */
  int          fi_maxsecno;
        /* Maximum number of sectors */
```

```
        short        fi_secptrk;
              /* Sectors per track (unused) */
        short        fi_trkpcyl;
              /* Tracks per cylinder (unused) */
        int          fi_secpcyl;
              /* Sectors per cylinder (unused) */
        int          fi_ncyl;
              /* Number of cylinders (unused) */

        caddr_t      fi_realccw;
              /* Real address of the ccw's */
        ccw_t        fi_deccw;
              /* Define extent ccw */
        ccw_t        fi_locccw;
              /* Locate ccw */
        ccw_t        fi_rwccw;
              /* Read/write ccw */
        delist_t     fi_delst;
              /* Define extent list */
        loclist_t    fi_loclst;
              /* Locate list */
        caddr_t      fi_idaws -MAX_FBA_COUNT / NBIDA +1 -;
              /* IDA words */

        unsigned int fi_nsiord;
              /* number of read sio */
        unsigned int fi_nsiowr;
              /* number of write sio */
        unsigned int fi_nblksrd;
              /* number of blocks read */
        unsigned int fi_nblkswr;
              /* number of blocks written */
        unsigned int fi_nretry;
              /* number of retries */
        unsigned int fi_tsio;
              /* rmtime for last sio */
        double   fi_tottsio;
              /* microsecs from sio to intr */
      };
```

HDIORST Reread the VTOC minidisk partition table.

```
ioctl(fd, HDIORST, arg);
caddr_t arg;
```

The argument is not used. This command only works on the whole partition maintenance minidisk.

Related Information
In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "open, openx, creat" in topic 1.2.199, "read, readv, readx" in topic 1.2.224, and "write, writex" in topic 1.2.330.

2.5.9 fd

Purpose
Supports the diskette device driver.

Synopsis
#include <sys/devinfo.h>

Description
The **fd** special file is unique to AIX PS/2. The diskette special file provides block and character (raw) access to diskettes in the diskette drives, allowing only one process to have a diskette drive open for writing at a time. Removing the diskette from the drive with diskette files still open may cause various I/O system calls to return errors.

The minor device number specifies both the driver number and the format of the diskette to be read or written. Following are the special file names used to access the diskette drives containing the specified diskettes (formatted).

Special File	Drive	Media
/dev/fd0	A	3.5 inch 1.44 Megabyte
/dev/fd1	B	3.5 inch 1.44 Megabyte
/dev/fd0h	A	3.5 inch 1.44 Megabyte
/dev/fd1h	B	3.5 inch 1.44 Megabyte
/dev/fd0l	A	3.5 inch 720 Kilobyte
/dev/fd1l	B	3.5 inch 720 Kilobyte
/dev/fd1x	External	5.25 inch 360 Kilobyte
/dev/fd3	External	5.25 inch 1.2 Megabyte

The special file names **/dev/rfd0, /dev/rfd1**, and so on, refer to the character (raw) interface to the diskette drives.

Warning: Accessing a diskette drive with a special file name that does not correspond to the format of the inserted media may lead to loss of data.

Subtopics

2.5.9.1 ioctl Operations

The **IOCTYPE** type **ioctl** system call returns the device type **DD_DISK**, defined in the **sys/devinfo.h** header file.

The **IOCINFO** type **ioctl** system call returns the following structure, defined in the **sys/devinfo.h** header file:

```
struct devinfo {
    char  devtype;
    char  flags;
    union {
       struct {            /* for disks */
          short  bytpsec;  /* bytes per sector */
          short  secptrk;  /* sectors per track */
          short  trkpcyl;  /* tracks per cylinder */
          long   numblks;  /* number of blocks on diskette */
       } dk;
       ...              /* for other devices */
    } un;
};

 /* flags */
#define DF_FIXED 01  /* non-removable */
#define DF_RAND 02   /* random access possible */
#define DF_FAST 04   /* a relative term */
```

2.5.9.2 Error Messages

The error messages printed out by the **fd** driver are of the following form:

```
fd_err_log: TTTT error on dev (MAJ/MIN), blkno=BBBBB
fd_err_log: status: ST UN S0 S1 S2
```

Where:

TTTT Is either **read** or **write**

MAJ Is the major device number

MIN Is the minor device number

BBBBB Is the disk block (sector) number.

The **status** bits have the following definitions:

ST (Internal state at time of error):

3 => error occurred during a Recalibrate operation

4 => error occurred during a Seek operation

6 => error occurred during a Read or Write operation

UN (The drive unit number. This number should be 0 for the first drive and 1 for the second drive).

S0 has the following bit definitions:

c0 Interrupt code mask

00 Normal Termination

40 Abnormal Termination

80 Invalid Command

c0 Ready signal changed

20 Seek command complete

10 Fault signal received from drive

08 Not ready

04 Current state of the head select line

03 Unit select

S1 has the following bit definitions:

80 Attempt to access sector beyond end of cyl

40 Not used

20 CRC error in the ID or data field

10 Host did not service FDC fast enough

08 Not used

04 Specified sector not found

02 Attempt to write a write protected disk

01 Missing address mark

S2 has the following bit definitions:

80 Not used

40 Control mark

20 CRC in the data field

10 Wrong Cylinder

08 Scan command: equal condition satisfied

04 Scan command: sector not found

02 Bad Cylinder

01 Missing Data address mark

Files

/dev/fd0, /dev/fd1, ...
/dev/rfd0, /dev/rfd1, ...

Related Information
In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137 and "fs" in topic 2.3.20.

2.5.10 hd

Purpose
Supports the fixed-disk device driver.

Synopsis
#include <sys/devinfo.h>

Description
The fixed-disk device driver provides block and character (raw) access to minidisks on the fixed-disk drives. The **hd** special file is unique to AIX PS/2. The system determines the association of the minor device number to the minidisk. Normally, the special files **/dev/hdn** and **/dev/rhdn** are given the minor device number **n**.

In raw I/O, the buffer must always begin on a 512 byte boundary, and counts must be a multiple of 512 bytes (a physical disk block). Likewise, **lseek** system calls must specify a multiple of 512 bytes. However, for the most efficient raw I/O, the buffer should be on a 4096 byte boundary and counts should be a multiple of 4096 (the page size).

Subtopics
2.5.10.1 ioctl Operations

2.5.10.1 ioctl Operations

The **IOCTYPE** type **ioctl** call returns the value **DD_DISK**, defined in **sys/devinfo.h**.

The **IOCINFO** type **ioctl** call returns the following structure, defined in **sys/devinfo.h**:

```
struct devinfo {
   char devtype;
   char flags;
   union {
      struct {        /* for disks */
      short bytpsec;  /* bytes per sector */
      short secptrk;  /* sectors per track */
      short trkpcyl;  /* tracks per cylinder */
      long numblks;   /* blocks this mini-disk */
     } dk;
       ...                    /* for other devices */
  } un;
};

/*flags */
#define DF_FIXED 01  /* non-removable */
#define DF_RAND  02  /* random access possible */
#define DF_FAST  04  /* a relative term */
```

Files

/dev/hd0, /dev/hd1,...
/dev/rhd0, /dev/rhd1,...

Related Information
In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "lseek" in topic 1.2.161, and "fs" in topic 2.3.20.

2.5.11 hft

Purpose
Implements a high-function virtual terminal device.

Synopsis

#include <sys/hft.h>

Introduction
The **hft** device driver allows one or more virtual terminals to be opened and used by application processes. The **hft** device driver is unique to AIX PS/2. Only one of these virtual terminals can be active at any given time; any others are hidden from view. The operator uses keyboard "hot key" sequences to switch between the active and inactive ones. There is an AIX command called **open** to create new virtual terminals from the keyboard.

Each of the virtual terminals can be set to specific terminal characteristics. There are AIX commands such as **display**, **keyboard**, **locator**, **sound**, and **stty** which can be used to adjust these characteristics.

This section describes the programmer's interface to **hft**. This interface was used by IBM to program the AIX commands mentioned above, and is also available for you to use in your application programs.

Getting Started
The **/usr/lib/samples** directory contains several **README** files that describe more information of interest to the programmer. There is also a **/usr/lib/samples/hft** directory which contains many example programs which show how to program the various **hft** functions. These examples often serve as a good starting point for constructing your own programs. If you are not familiar with **hft**, then you should first read the **Concepts** below, then proceed by trying one or more of the example programs. You should also print out a copy of **/usr/include/sys/hft.h** which will help in understanding the structures described in this section.

Concepts

The virtual terminal concept supports the illusion that more devices exist than are physically present and that these virtual devices have characteristics and features not necessarily limited by the actual physical devices. In addition to displays and keyboards, virtual terminals support locators and sound generators. Virtual terminals are logically independent of each other but share physical resources over time. Each virtual terminal embodies the characteristics of a single keyboard send/receive terminal. That is, it recognizes and processes the data stream received from the process causing the requested actions to occur, for example, move the cursor, or draw characters onto the virtual display, or change the attributes of characters. In addition to these actions, the outbound data stream can cause the generation of sequences of continuous tone sounds or cause the virtual display to be rendered on any of the available physical displays.

A virtual terminal receives input from a virtual keyboard and/or a virtual locator; it outputs to a virtual display. Thus the virtual terminal can always expect to get input from its virtual input devices and can always output to its virtual display. Since physical devices are shared between virtual terminals, there will be times when a process desires to read or

write and the physical device is allocated to another virtual terminal. In these cases the process is blocked until the device becomes allocated. The virtual terminal that can accept physical input or modify the physical screen at a given time is called the **active** virtual terminal. Processes are never blocked for write if they are sending a data stream in KSR mode. The **inactive** virtual terminal accepts the data stream and outputs it to a presentation space. This presentation space will be rendered on the physical display when the virtual terminal becomes active.

There is a Screen Manager which performs the allocation of physical devices to virtual terminals over time. The Screen Manager implements a ring of virtual terminals in which one virtual terminal is active and the others are inactive. There are keyboard "hot key" sequences to control switching between the active and inactive virtual terminals. The Screen Manager allocates needed physical devices to the virtual devices of the active virtual terminal. The Screen Manager also allows the switching of virtual terminals to occur under program control.

There is a Resource Manager that allows global changes to be made to the default characteristics of new virtual terminals. For instance, it is possible to redefine the default display so that subsequent opens of the **hft** device will have different characteristics.

Each virtual terminal provides a model of a single terminal that can be in one of the following modes at a given time:

Keyboard Send-Receive Mode (KSR

Monitor Mode (MOM)

The KSR mode emulates an ASCII terminal using a data stream, which is described in detail in "data stream" in topic 2.4.3. The monitor mode allows applications to have a direct output path to the display hardware and shortened path for keyboard and locator. The form of the data accepted in each mode is unique to that mode. This optimizes the movement of data between the virtual terminal and the application program and supports the different functions within each mode. The default mode is KSR, which supports existing applications expecting an ASCII terminal.

Additional functions supported include:

Reporting data from input devices such as locators (mice
Switching between interactive and noninteractive state
Changing color palette setting
Controlling the sound hardwar
Switching between the monitor mode and KSR mode

The virtual terminal supplies default values for keyboard-to-character mapping, character-to-display mapping, echo/break specification, tab rack, and protocol mode flags to be used until a definition is received from the application.

This **hft** facility is the kernel-level support for virtual terminals. Since the association of virtual terminals to physical terminals is dynamic, this special file, which represents the physical terminal, is multiplexed across virtual terminals by expanding the **open**, **close**, **read**, **write**, and especially the **ioctl** system calls to the driver. This type of driver is specified by the **M** flag in the master file. Many extra **ioctl** system calls are provided to allow access to advanced features of the **hft** facility. The facilities described in "termio" in topic 2.5.28 also apply

to the virtual terminal.

The first (or only) **hft** is minor device 0, and special file **/dev/hft** is associated with it. The special file **/dev/console** is minor device 1.

Each time **/dev/hft** is opened, a new **hft** virtual terminal is created and opened. A maximum of 16 virtual terminals can be opened due to limits on system resources. Each **hft** virtual terminal is given both a channel number and an I/O device number (IODN). You use the channel number if you wish to reopen an already existing virtual terminal. You use the IODN as part of certain **hft ioctl** operations, such as controlling the screen manager. The IODN specifies upon which virtual terminal to operate.

To reopen an existing virtual terminal, open the special file **/dev/hft/i**, where **i** is the number of an open driver channel. The channel number can be determined with the HFGCHAN **ioctl** operation. The **/dev/console** special file is channel number 1. The **/dev/tty** special file is often used by programs to direct output and input to the current controlling terminal. For AIX, this special file acts as a synonym for the current virtual terminal.

A process can also communicate with the **hft** screen manager by opening the **/dev/hft/mgr** file. Only the screen manager **HFQSMGR** and **HFCSMGR ioctl** operations can be issued to this file. **read** and **write** system calls are not allowed.

The **/usr/lib/samples/hft** directory contains sample programs that use the **hft** virtual terminal subsystem. See the file **/usr/lib/samples/README.hft** for more information about these sample programs. When you develop programs that access the **hft** device driver, you should use the header file **hft.h** to define the necessary structures and constant definitions.

Subtopics

2.5.11.1 Contents of hft Section

2.5.11.2 Open/Close

Refer to the sample program **hftopen.c** for an example of the topics discussed in this section.

Subtopics

2.5.11.2.1 Creating a New Virtual Terminal

The **hft** device driver is a multiplexed device where each virtual terminal has a specific **hft** channel number associated with it. Opening the file **/dev/hft** causes the Screen Manager to create a new virtual terminal using the first available **hft** channel.

The **hft** channels available are:

Channel Number	**Device Name**	**Use**
0	Reserved	Screen Manager
1	**/dev/hft/1**	console (/dev/console)
2	Reserved	
3	**/dev/hft/3**	Virtual Terminal
4	**/dev/hft/4**	Virtual Terminal
.		
.		
.		
17	**/dev/hft/17**	Virtual Terminal

The function call:

```
fildes = open("/dev/hft",O_RDWR)
```

causes the Screen Manager to create a new virtual terminal and assign the first available channel to it. The monitor immediately switches to the new screen. The file descriptor returned can be used to read or write from that virtual terminal.

2.5.11.2.2 Determining the New Terminal's Channel Number

Some **hft** operations require the channel number of the terminal as a parameter. After opening a new terminal issue the function call:

```
channel = ioctl(fildes,HFGCHAN,0)
```

Note that this function actually returns the channel of whichever terminal is currently displayed. Since the Screen Manager switched you to the new terminal immediately after creating it, the channel number of the new virtual terminal is returned.

2.5.11.2.3 Redirecting Input and Output

After you open a new virtual terminal, the standard input is still from the original virtual terminal (usually **/dev/console**) and the output is also sent to the original virtual terminal, even when this virtual terminal is not displayed. To redirect input and output to the new virtual terminal:

```
outdesc = freopen("/dev/hft/(channel number)","w",stdout)
indesc = freopen("/dev/hft/(channel number)","r",stdin)
```

Now you can use **printf**, **getc**, and other functions normally. When the original console screen is displayed, keyboard input is sent to the program running on the console and output is sent to the console screen. When the new virtual terminal screen is displayed, keyboard input is sent to the program running on the virtual terminal and output is sent from that program to the virtual terminal screen.

2.5.11.2.4 Switching between Virtual Terminals

The operator uses **Alt-Action** or **Shift-Action** to switch between the virtual terminals available for display. Each time these key combinations are pressed, the next or previous virtual terminal is displayed.

Your program can also control the switch between virtual terminals by issuing the **HFQSMGR** and **HFCSMGR ioctl**s to the **hft** Screen Manager using pseudo device **/dev/hft/mgr**. Refer to the sample programs **hftcsm.c** and **hftqsm.c**.

2.5.11.3 Input

Data read from an **hft** device with the **read** system call can contain not only character data entered from the keyboard, but also input from other devices, such as a locator or mouse.

Mouse input arrives in the form of Mouse Reports. See the section called "Using the Mouse" in topic 2.5.11.3.1 for more information.

Keyboard input normally arrives in the form of ASCII characters and control sequences as described in "data stream" in topic 2.4.3. Such characters are essentially the same as the datastream that an ASCII terminal would produce.

If an application needs to know the exact state of the keyboard, such as when keys are pressed and released then Untranslated Key Control should be used. This is done by turning off the **HFXLATKBD** protocol. See "Set Protocol Modes" in topic 2.5.11.7.6. When keyboard input is untranslated the **hfunxlate** sequence is returned for each change in keyboard state. See "keyboard" in topic 2.5.13 for additional information.

The key position identifies the logical key pressed. The key status bits indicate **Alt**, **Alt-Gr**, **Ctrl**, **Shift**, **Caps Lock**, and **Num Lock** key states.

Note: This control sequence contains binary data. To prevent the binary data from being misinterpreted as ASCII control codes, set the terminal's canonical processing off.

The structure of the untranslated key control is:

```
struct hfunxlate
{
    char hf_esc;
    char hf_lbr;
    char hf_ww;
    char hf_keypos;
    char hf_scancode;
    char hf_status[2];
};
```

The fields of the structure are:

Field	Description
hf_esc	ESC (0x1B)
hf_lbr	**[** (0x5B)
hf_ww	**w** (0x77)
hf_keypos	Key Position

Field	Description
hf_scancode	Scan Code (See *PS/2 Hardware Interface Technical Reference*.) The scancode returned is the "makecode" as defined in Set 3 scancode tables.
hf_status[0]	Status:

HFUXSHIFT	A shift key is pressed.
HFUXCTRL	**Ctrl** key is pressed.
HFUXALT	**Alt** key is pressed.
HFUXCAPS	**Caps Lock** mode is in effect.
HFUXNUM	**Num Lock** mode is in effect.
HFUXMAKE	If set, key has been pressed. If not set, key has been released.

hf_status[1] Status:

HFUXRPT	Automatic repeat (typematic) state.
HFUXLSH	Left shift state.
HFUXRSH	Right shift state.
HFUXLALT	Left alternate shift state
HFUXRALT	Right alternate shift state. (**Alt-Gr** for 102-key keyboards)

Subtopics
2.5.11.3.1 Using the Mouse

2.5.11.3.1 Using the Mouse

Refer to the **hftmouse.c** sample program for an example of the topics discussed in this section.

AIX architecture allows many forms of non-keyboard input, including a class called **locator**. Locators can provide input in either relative or absolute coordinates. The PS/2 mouse produces Mouse Reports which contain relative coordinates.

Enable/disable mouse:

Use the **HFLOCATOR** bit in the **hfprotocol** structure. See "Set Protocol Modes" in topic 2.5.11.7.6. Note that the mouse is initially set to disabled.

Input Device Report:

Mouse data is reported in the **hflocator** structure, in response to the **read** system call.

Mouse Report

hf_esc ESC (0x1B)

hf_lbr **[** (0x5B)

hf_why **y** (0x79)

hf_deltax The X delta, a signed integer that holds the relative X delta accumulations in counts of 0.25 millimeters of the locator movement in twos-complement form. This information is sent to the virtual terminal to indicate horizontal movement since the last locator movement.

hf_deltay The Y delta, a signed integer that holds the relative Y delta accumulations in counts of 0.25 millimeters of the locator movement in twos-complement form. This information is sent to the virtual terminal to indicate vertical movement since the last locator movement.

hf_seconds Time of the locator report in whole seconds since system startup.

hf_sixtyths The fractional part, of locator report time stamp, in 1/60th seconds.

hf_buttons The status of the locator buttons. This information is sent to the virtual terminal to indicate a change in the status of the buttons since the last locator movement in the following manner:

HFBUTTON1 Button 1 has been pressed.
HFBUTTON2 Button 2 has been pressed.

hf_stype 0

Note: This control sequence contains binary data. To prevent the binary data from being misrepresented as ASCII control codes, set the terminal's canonical processing off. See ICANON in topic 2.5.28 in "termio" for details.

Query Locator:

The resolution of the locator and the current threshold settings can be obtained by issuing the **Query Locator** command. This is described within the section on the **Query ioctl**.

Change Locator Sample Rate:

The sampling rate of the locator can be changed by the **HFCHGLOCRATE** option of the **Reconfigure ioctl**.

Set Locator Thresholds:

The locator device receives notice of horizontal and vertical movement. The delta of these movement events are monitored by the driver, until the accumulated events exceed either the horizontal or vertical thresholds, or both. The locator device accumulates measurements at consecutive samplings. When a threshold is exceeded, the driver queues the information to the virtual terminal. When the status of the locator buttons change, the accumulated measurements are returned to the virtual terminal, even if these measurements do not exceed a threshold. The virtual terminal provides neither echoing nor positional management functions for the locator.

Each opened virtual terminal has its own threshold values. When a virtual terminal is opened, the threshold values default to 2.75 millimeters horizontal and 5.5 millimeters vertical. If the thresholds are 0, each event report is returned to the virtual terminal at the sampling rate supported by the locator device driver.

Setting the **HFLOCATOR** bit to 0 in the protocol mode definition or setting both thresholds to the maximum values completely disables the locator input. Setting a -1 for either or both threshold values indicates not to change the current setting.

The **hfloth** structure is used for the locator threshold command, and it contains the following fields:

Field	**Value**
hf_intro.hf_typehi	**HFLOTHCH**
hf_intro.hf_typelo	**HFLOTHCL**
hf_sublen	2

Field	**Value**
hf_subtype	1
hf_hthresh	Specifies the horizontal threshold in values from 0 to 32767 in units of 0.25 millimeters.
hf_vthresh	Specifies the vertical threshold in values from 0 to 32767 in units of 0.25 millimeters.

2.5.11.4 Output

The virtual terminal is initially set into KSR mode which emulates an ASCII display. If an application needs direct control of physical hardware, then it can enter monitor mode.

Subtopics

2.5.11.4.1 Keyboard Send-Receive Mode (KSR)

In KSR mode, each byte written to the virtual terminal is interpreted as an ASCII code, which can be a displayable character, a single-byte control, or part of an escape or control sequence. "data stream" in topic 2.4.3 explains the supported ASCII/ANSI data stream in detail. KSR mode also supports a number of special control sequences specific to the virtual terminal environment.

A KSR virtual terminal has a presentation space (PS) of a fixed number of columns per line, and a fixed number of lines. A symbol can be placed at any column on any line in the presentation space. A pointer into the virtual terminal defines the cursor position with a column and a line number. Graphics from the KSR data stream are placed in the PS relative to the cursor position. Keyboard input also relates to the cursor position.

Two common modes for displaying graphics are replace and insert. In replace mode, a graphic character sent to a KSR terminal is placed above the cursor, replacing the symbol already there. In insert mode, a graphic character sent to a KSR terminal is also placed above the cursor, but the symbol above the cursor and all symbols to the right on the same line are shifted right one column position on the line. Characters shifted from the last column on the line disappear.

Another mode determines cursor movement after the last column position of a line. This mode, automatic new line (AUTONL), determines if the cursor wraps around to the first column position of the next line or stays at the last column on the current line.

If AUTONL is set, the cursor moves to the first column position of the following line. If the cursor happens to be on the bottom line of the presentation space, the presentation space scrolls up one line. If AUTONL is reset, the cursor stays on the last column of the current line.

Blank lines in the presentation space and erased character positions display in the active background color with normal attributes.

To set the KSR protocol modes, write a protocol mode control, which is described under "Set Protocol Modes" in topic 2.5.11.7.6. Specify the type as **HFKSRPROH**, **HFKSRPROL**.

2.5.11.4.2 Monitor Mode (MOM)

Refer to the **hftmom.c** sample program for an example of the topics discussed in this section.

Programs that choose to interact more efficiently with a virtual terminal or that must operate the display in all-points-addressable mode should select the monitor mode of the virtual terminal. In this mode, the program performs output directly to the display adapter via a memory mapped I/O bus, thus avoiding write system calls. Such a program can optionally read data from a circular buffer, (known as the "Input Ring Buffer"), thus avoiding **read** system calls. Some execution speed is gained by operating in this mode, but portability is sacrificed because the program depends on specific display adapters.

Notes:

1. Do not leave terminal open in monitor mode.
2. No more than 1 process should be open to a virtual terminal that is in monitor mode.

In order for a user program to switch from normal KSR mode to monitor mode, it must perform several mode changes, which are accomplished using system calls. The display-sharing concept using virtual terminals causes the program in monitor mode to participate in the **next window** function by temporarily releasing the display. This is also accomplished using system calls. While the user program is active to the display, it performs output operations directly to the display hardware with memory mapped I/O ports.

Entering Monitor Mode:

The first mode change the user program should perform is to issue the **HFSMON ioctl** operation to enable monitor mode signals **SIGGRANT** and **SIGRETRACT**, and to specify the method by which processes are to receive the signals. (See "Enter Monitor Mode (HFSMON)" in topic 2.5.11.5.2.)

Next, the program should write a protocol mode control, which is described under "Set Protocol Modes" in topic 2.5.11.7.6, specifying the type **HFMOMPROH**, **HFMOMPROL**. Only certain protocols are valid for monitor mode.

The virtual terminal is now in monitor mode.

Only certain controls are valid for the **write** system call while in monitor mode. All other ASCII codes and controls are ignored. The valid controls and VTDs are:

Disable Manual Input (DMI
Enable Manual Input (EMI
Set Keyboard LED
Set Locator Threshol
Soun
Cancel Soun
KSR Protocol (Enable/Disable Locator
MOM Protoco
Screen Request (Establish Ring Buffer
Screen Release

Screen Request and Input Ring Buffer Definition:

Although the virtual terminal is in monitor mode, the program can perform direct operations on the display hardware only when granted permission by the operating system. The program first writes a screen request Virtual Terminal Command.

This request uses the **hfmomscreq** structure, which contains the following fields:

Field	Value
hf_intro.hf_len	The length of the request from the start of this **len** field up to and including the ring buffer.
hf_intro.hf_typehi	**HFMOMREQH**
hf_intro.hf_typelo	**HFMOMREQL**
hf_sublen	Subheader length
hf_subtype	Subheader type
hf_ringlen[2]	Shows the length of the **hfmomring** structure in bytes.
hf_ringoffset[4]	Shows the offset to the input buffer ring (offset from the **hf_ringlen** field).

The **hf_ringlen** field specifies the size of the structure including the pointers and status fields. The program can directly access input key and locator data contained in the buffer without issuing **read** system calls. A minimum recommended ring buffer size would be 32 bytes.

The ring buffer structure (**hfmomring**, defined following) can be at any location in memory aligned on a word boundary. **hf_ringoffset** is the difference between the ring buffer address and the address of **hf_ringlen**, and it must be a positive value. Usually, the **hfmomring** ring buffer structure is defined so that it immediately follows the **hfmomscreq** structure in memory. Note that the compiler may implicitly insert one or more filler bytes between the two structures to align them at a memory address boundary. The value of **hf_ringoffset** must reflect such filler bytes. See the **/usr/lib/samples/hft/hftmom.c** source file for an example of how to calculate **hf_ringoffset**.

If you do not want to specify or use a ring buffer, then set the **hf_len** field of the **hf_intro** to the size of only the introducer. In this case, read input with the standard **read** system call. The **hftmom2.c** sample program shows how to develop a program without use of the ring.

```
struct hfmomring
{
   char hf_rsvd[2];
   char hf_intreq;
   char hf_ovflow;
   unsigned hf_source;
   unsigned hf_sink;
   int  hf_unused[5];
   char hf_rdata[HFRDATA];
};
```

The fields in this structure are defined as:

Field	Value
hf_rsvd	Reserved.
hf_intreq	Interrupt request can be set to 0xFF by the application to cause the virtual terminal subsystem to send a **SIGMSG** signal each time an input event occurs. If this flag is set to 0 (the default), then a signal is sent to the application only when the buffer goes from being empty to nonempty. This byte is automatically reset to 0 by the virtual terminal each time it stores input data into the ring buffer. See "Reading Input Data from the Ring Buffer" for further discussion.
hf_ovflow	Overflow determines whether the input buffer ring can accommodate more input information. A value of 0xFF indicates an overflow; 0x00 indicates normal operation.
hf_source	Ring offset for virtual terminal represents the offset into the input ring where the virtual terminal queues keyboard and locator input. This offset starts from the beginning of the ring, so the absolute minimum value for the virtual terminal offset is 32 bytes. Application programs must not alter this field. If a program attempts to alter it, then the virtual terminal is killed. See "Reading Input Data from the Ring Buffer" for further discussion.
hf_sink	Ring offset for application shows the offset into the input ring from which the application reads keyboard and locator information from the event queue. This offset also starts from the beginning of the input ring, so the minimum value for this offset is 32 bytes. See "Reading Input Data from the Ring Buffer" for further discussion.
hf_unused	Reserved.

Reading Input Data from the Ring Buffer:

The program should initialize the offsets **hf_source** and **hf_sink** to be equal. This indicates buffer empty condition. The program should then issue the **pause** system call, waiting for input. When the buffer goes from being empty to not empty, the program receives a **SIGMSG** signal. (Note that sending the **hfmomscreq** structure and defining the input ring buffer enables the sending of this signal.) The program should extract characters from the ring buffer while incrementing the **hf_sink** offset for each character extracted, making sure to wrap around after reaching the end of the buffer. Care should be taken to ensure the buffer empty condition is properly detected. The program should test the equality of the offsets after it has updated the **hf_sink** offset. Therefore, the order of operation is: extract a character, update the offset in its memory location, and test the equality of offsets; if the offsets are equal, then set **hf_intreq** to 0xFF.

If **hf_source == hf_sink - 1** (module ring size), then the ring buffer is full. If **hf_ovflow == 0xff**, then an overflow condition exists. The overflow condition indicates input data has been lost. The program resets the overflow condition by clearing **hf_ovflow**.

Certain keys can be designated so they can be obtained using the **read** system call. This is particularly useful when such keys are the **Intr** and **Quit** keys (see "termio" in topic 2.5.28). These keys are designated using HFSECHO. Thus, by designating these keys in the break map, and by setting the ISIG mode of **termio**, it is possible to asynchronously interrupt a monitor mode program by pressing one of these keys.

Next Window Function:

If a virtual terminal in monitor mode is active, pressing the Next Window key causes a **SIGRETRACT** signal to be sent to the process or group of processes specified by the HFSMON type **ioctl** system call. Before activating the next virtual terminal, the operating system allows the program a chance to save the state of the display hardware, such as registers and refresh memory. After this is done, the program should write a screen release control to the terminal to inform the operating system the state of the display hardware can be changed.

The screen release control is given by the **hfmomscrel** structure:

Field	**Value**
hf_intro.hf_len	The length of the entire structure, minus 3; typically 6 bytes.
hf_intro.hf_typehi	**HFMOMRELH**
hf_intro.hf_typelo	**HFMOMRELL**

After the display is released, the next virtual terminal is activated. If this is not done within 30 seconds of the receipt of the **SIGRETRACT** signal, all processes in that terminal group receive a **SIGKILL** signal. This is a safeguard to prevent runaway monitor mode programs from disrupting the next window function.

The program can issue a **pause** system call if there is no work to do while the display is not available. When the monitor mode virtual terminal is activated again with the Next Window key, the program receives a **SIGGRANT** signal. In other words, the program can resume direct output to the display. The display hardware state cannot be assumed to be the same as when the program released it. Therefore, the **SIGGRANT** signal handler must restore the state that the **SIGRETRACT** signal handler saved.

Exiting Monitor Mode:

When the program has no further use of the monitor mode, it should first write a screen release control, followed by a KSR protocol control. The screen release control causes addressability to video RAM and the I/O ports to be rescinded. This is especially important if the virtual terminal is open by another process, such as the parent process, which is often the command shell. Changing back to KSR protocol tells the **hft** device driver that it may resume having control over the hardware. Only in KSR mode will the **hft** driver draw characters on the screen. If the program is certain that no other processes have the terminal open, it can simply issue a **close** system call to remove that virtual terminal.

Next, an **HFCMON ioctl** operation should be issued to turn off monitor mode signalling to this process or other process in the terminal group.

Signals:

In addition to the standard terminal signals (**SIGINT** and **SIGQUIT**), the virtual terminal generates other unique signals to inform the application program of asynchronous events. These signals include:

SIGGRANT Informs the user program that the display hardware can be directly accessed. This signal is sent following a monitor mode screen request VTD sequence. It is also sent after a monitor mode terminal has been made active with the next window key.

SIGRETRACT Informs the user program that the display hardware must be released for use by another program. This signal is sent after a monitor terminal being made inactive with the next window key.

SIGKILL Sent to all processes in the terminal **tty** group to enforce the **SIGRETRACT** signal. If the user program does not respond with a screen release VTD sequence within 30 second after receiving a **SIGRETRACT** signal, the **SIGKILL** terminates all processes associated with that virtual terminal and the terminal is closed.

SIGMSG Informs the user program that data has been placed into a previously empty input buffer.

2.5.11.4.3 Controlling Sound through the Speaker

Refer to the sample programs, **hftsound.c**, **hftcansnd.c**, **hftdsnd.c**, and **hftesnd.c** for examples of the topics discussed in this section.

Sound:

This command sends output to the speaker. The mode byte determines whether to execute sound commands for the active virtual terminal and whether to interrupt the application after the sound command executes. No range check is made for the frequency or duration values. The **hfsound** structure is used for this command:

Field	Value
hf_intro.hf_typehi	**HFSOUNDCH**
hf_intro.hf_typelo	**HFSOUNDCL**
hf_sublen	Subheader length
hf_subtype	Subheader type
hf_mode	Mode: **HFSIGSOUND** If set, causes the **SIGSOUND** signal to be sent to the process when this sound command is executed or discarded. If not set, then no signal is sent. **HFEXECALWAYS** If set, causes this sound command to be executed whether or not this virtual terminal is active. If not set, then the sound command is executed only if the terminal is active; if not active, the sound command is discarded.
hf_dur	Duration in 1/128 seconds.
hf_freq	Frequency in hertz.

Enable Sound Signal (HFESOUND):

This command informs the terminal driver of the intent to use sound, enabling the routing of the sound response signal. This is invoked by the following:

```
int ioctl(fildes, HFESOUND, arg)
int fildes;
struct hfsmon *arg;

struct hfsmon
{
  int hf_momflags;
};
```

The **hf_momflags** field contains one of the following values:

HFSINGLE Only the process issuing the **ioctl** system call is to receive a

sound response signal.

HFGROUP All members of the current process group are to receive a sound response signal.

Disable Sound Signal (HFDSOUND):

This informs the terminal driver of the intent to discontinue the use of sound. Sound response signals are not sent. This is invoked by the following:

```
int ioctl (fildes, HFDSOUND, 0)
int fildes;
```

Cancel Sound:

The cancel sound command removes all commands from the speaker device that do not want sound commands executed. Only the commands that have the **HFEXECALWAYS** flag are left in the active terminal queue. An inactive terminal ignores this command.

Sending a **cancel** and/or **enable sound** command flushes the speaker driver queue when a virtual terminal transition occurs. Regardless of whether the sound request is executed or purged, the virtual terminal receives a response (**SIGSOUND**) if the response flag is set (bit 0 of sound command byte 0 is equal to 1).

The **hfcansnd** structure is used for this command, and it contains the following fields:

Field	Value
hf_intro.hf_typehi	**HFCANSNDCH**
hf_intro.hf_typelo	**HFCANSNDCL**

2.5.11.5 ioctl Operations

The **hft** supports a number of operations issued by the **ioctl** system call to provide access to features of the **hft**. See "ioctlx, ioctl, gtty, stty" in topic 1.2.137 for details about the syntax of the system call itself. For information about issuing requests for these operations to an emulated **hft** device, see "Considerations for hft Emulation" in topic 2.5.11.11. There are also other operations which control the Virtual Terminal through use of writes. See "Virtual Terminal Commands" in topic 2.5.11.7.

Subtopics
2.5.11.5.1 Query I/O Error (HFQEIO)
2.5.11.5.2 Enter Monitor Mode (HFSMON)
2.5.11.5.3 Exit Monitor Mode (HFCMON)
2.5.11.5.4 Get Virtual Terminal ID (HFGETID)
2.5.11.5.5 Get Channel Number (HFGCHAN)
2.5.11.5.6 Query (HFQUERY)

2.5.11.5.1 Query I/O Error (HFQEIO)

If an I/O operation or other system call to the **hft** fails due to a hardware error, the system call returns a nonzero value and sets the **errno** external variable to the value **EIO**. The calling program can get a more detailed device error code by using **ioctl** to issue an HFQEIO operation. This is invoked by the following:

```
int ioctl(fildes, HFQEIO, 0)
int fildes;
```

The return value from the HFQEIO **ioctl** operation is either 0 (indicating that the last I/O operation was successful), -1 (indicating that the HFQEIO operation itself failed), or the error code for the last **hft** I/O operation.

The possible terminal error codes are:

6401 Invalid virtual terminal IODN

6461 Maximum number of virtual terminals open

6480 Invalid operation

6516 Invalid virtual address

6521 Unsuccessful, invalid length specified in VTD block

6522 Unsuccessful, invalid major type

6523 Unsuccessful, invalid minor data

6524 Unsuccessful, invalid minor type

6527 Unsuccessful, VTD block exceeds 128K bytes

6528 Unsuccessful, VTD block is less than the minimum length

6531 Unsuccessful, cannot remap a character set other than unique 1 or 2

6532 Invalid locator/mouse/tablet type request

6533 Unsuccessful, invalid font ID

6537 Invalid graphics asynchronous device driver request

6538 Specified device not configured

6539 Specified device not selected

6544 Unsuccessful, data received for an inactive mode

6545 Unsuccessful, specified virtual terminal not active.

6546 Unsuccessful, invalid virtual terminal identifier

6548 Unsuccessful, invalid coordinates specified in Query ASCII Codes command

6549 Unsuccessful, invalid parameter detected in a control sequence

6550 Unsuccessful, unsupported control sequence or code received

6555 Unsuccessful, sound error

6562 Unsuccessful, invalid echo map length

6564 Unsuccessful, cannot remap keys reserved for resource controller

6565 Unsuccessful, invalid flags in the keyboard mapping structure

6566 Unsuccessful, invalid key position

2.5.11.5.2 Enter Monitor Mode (HFSMON)

This requests monitor mode. Monitor mode provides a program with direct control of the screen and keyboard. This is invoked by the following:

```
int ioctl(fildes, HFSMON, arg)
int fildes;
struct hfsmon *arg;

struct hfsmon
{
  int hf_momflags;
  int hf_momscnt;
  caddr_t hf_momsaddrs[MAX_MON_ADDRS];
};
```

The **hf_momflags** field contains one of the following values:

HFSINGLE Only the process issuing the **ioctl** system call is to receive monitor mode signals.

HFGROUP All members of the current process group are to receive monitor mode signals.

The **hf_momscnt** field must be set to **MAX_MON_ADDRS** to define the maximum number of entries in the **hf_momsaddrs** field.

The **hf_momsaddrs**[0] field will return the virtual address of the start of the video I/O buffer. For the VGA adapter, this address will correspond to 0xA0000 in PC memory. (See the *PS/2 Hardware Interface Technical Reference*.)

The other elements in **hf_momsaddrs** are reserved for adapters that work with more than one video address. See **/usr/lib/samples/README.mom** for information on other supported adapters.

2.5.11.5.3 Exit Monitor Mode (HFCMON)

Releases monitor mode. This is invoked by the following:

```
int ioctl(fildes, HFCMON, 0)
int fildes;
```

2.5.11.5.4 Get Virtual Terminal ID (HFGETID)

Gets identification information for the current **hft** virtual terminal. This is invoked by the following:

```
int ioctl(fildes, HFGETID, arg)
int fildes;
struct hfgetid *arg;

struct hfgetid  {
    unsigned hf_iodn;
    unsigned hf_pgrp;
    unsigned hf_chan;
};
```

The **hf_iodn** field is the I/O device number (IODN) of the virtual terminal. The **hf_pgrp** field is the process group ID; that is, the process ID of the terminal group leader. The **hf_chan** field is the channel number that is also returned by the HFGCHAN **ioctl** operation.

Each **hft** virtual terminal is given both a channel number and an IODN or I/O device number. You use the channel number if you wish to reopen an already existing virtual terminal. You use the IODN as part of certain **hft ioctl** operations, such as controlling the screen manager. The IODN specifies which virtual terminal is to be operated upon for certain **hft** ioctl operations such as controlling the screen manager. See the **hftgetid.c** sample program.

2.5.11.5.5 Get Channel Number (HFGCHAN)

Returns the current driver channel number as the value of the **ioctl** system call. This number can be used to open a specific virtual terminal. The **arg** parameter is ignored. This is invoked by the following:

```
int ioctl (fildes, HFGCHAN, 0)
int fildes;
```

See the **hftgchan.c** sample program.

2.5.11.5.6 Query (HFQUERY)

Refer to the **hftqdid.c**, **hftqpd.c**, **hftqpres.c**, and **hftqhft.c** sample programs for examples of the topics discussed in this section.

HFQUERY gets information about the current virtual terminal. This is invoked by the following:

```
int ioctl(fildes, HFQUERY, arg)
int fildes;
struct hfquery *arg;

struct hfquery  {
    char *hf_cmd;
    int hf_cmdlen;
    char *hf_resp;
    int hf_resplen;
};
```

The first two fields describe a buffer containing the command. The second two fields describe a buffer large enough to hold the largest possible response. Note that each command and response structure begins with a virtual terminal data (VTD) header. (See "Virtual Terminal Commands" in topic 2.5.11.7 for an explanation of the VTD header.) The following query commands use this **ioctl** operation.

Query Device IDs Command:

This command uses the **hfqdevidc** structure, which contains the following fields:

Field	Value
hf_intro.hf_typehi	**HFQDEVIDCH**
hf_intro.hf_typelo	**HFQDEVIDCL**

This command fills the response buffer with the information about the display devices. The information is returned in an **hfqdevidr** structure, which has the following fields:

Field	Value
hf_intro.hf_typehi	**HFQDEVIDRH**
hf_intro.hf_typelo	**HFQDEVIDRL**
hf_numdev	The number of devices for which data is reported.

The following fields are repeated for each physical device:

hf_devid	Physical device ID. The first device ID is the active display device ID, unless the **change physical display** command has changed the active display ID. The following values are possible: 0x0411**mmnn** VGA adapter with 8503 Display

0x0412**mmnn**	VGA adapter with 8512 Display
0x0413**mmnn**	VGA adapter with 8513 Display
0x0414**mmnn**	VGA adapter with 8514 Display
0x0415**mmnn**	VGA adapter with 8507 display
0x0416**mmnn**	VGA adapter with 8604 display
0x0418**mmnn**	8514/A adapter with 8503 display
0x0419**mmnn**	8514/A adapter with 8512 display
0x041A**mmnn**	8514/A adapter with 8513 display
0x041B**mmnn**	8514/A adapter with 8514 display
0x041C**mmnn**	8514/A adapter with 8507 display
0x041D**mmnn**	8514/A adapter with 8604 display
0x0421**mmnn**	reserved

Note: The **mm** value indicates whether the adapter is totally functional. When this value is 0x00, the adapter is totally functional. Any other value indicates the adapter is less than fully functional or not working at all, but is present on the machine. The **nn** value can be from 0x01 to 0x04 and differentiates between multiple instances of the same adapter type.

hf_class Display class (0x44).

Query Physical Device Command:

This command returns information about display or locator devices. The **hfqphdevc** structure is used to issue this command:

Field	**Value**
hf_intro.hf_typehi	**HFQPDEVCH**
hf_intro.hf_typelo	**HFQPDEVCL**
hf_phdevid	Physical device ID. The value 0 specifies the active device that is currently attached to the virtual terminal.

The response to this command gives the following information:

```
struct hfqphdevr
{
   char hf_intro[HFINTROSZ];
   char hf_sublen;
   char hf_subtype;
   /* locator  device */
   char hf_scale[4];
   char hf_locattr[1];
   char hf_rsvd[3];
   /* display device */
   char hf_attrib[4];
   char hf_pwidth[4];
   char hf_pheight[4];
   char hf_mwidth[4];
   char hf_mheight[4];
```

```
    char hf_bperpel[4];
    char hf_phdevid[4];
    /* display font */
    char hf_numfont[4];
    /* remainder is of variable length */
    /* struct hffont hffont[N];  where N is value in hf_numfont */
    char hf_fontstart;
     /* following is one color response */
     /* struct hfcolor hfcolor; */
 };

 struct hfqfont
 {
    char hf_fontid[4];
    char hf_fontstyle[4];
    char hf_fontattr[4];
    char hf_fontwidth[4];
    char hf_fontheight[4];
};

struct hfcolor
{
   char hf_numcolor[4];
   char hf_numactive[4];
   char hf_numfgrnd[4];
   char hf_numbgrnd[4];
   char hf_actcolor[4];
};
```

These structures are explained in the following sections that have headings beginning with the word **Physical**.

Physical Device Information VTD Header

Field	**Value**
hf_intro.hf_typehi	**HFQPDEVRH**
hf_intro.hf_typelo	**HFQPDEVRL**

Physical Locator Information

Field	**Value**
hf_scale	Scale factor (millimeters per 100 counts)
hf_locattr[0]	Locator attributes:
	HFLOCABS If set, then the locator device reports absolute coordinates (for example, a tablet device). If not set, then it reports relative coordinates (for example, a mouse).

Physical Display Device Information

Field	**Value**
hf_attrib[0]	Display device attributes:
	HFISAPA All-points-addressable (APA) display.

HFHASBLINK Blink function allowed.

All other values are reserved.

hf_attrib[2] Display device attributes:

HFHASCOLOR Color allowed.

All other values are reserved.

hf_attrib[3] Display device attributes:

HFCHGPALET Can change display adapter's color palette.

All other values are reserved.

hf_pwidth Displayable width of physical screen, expressed in pels or pixels for all displays.

hf_pheight Displayable height of physical screen, expressed in pels for all displays.

hf_mwidth Displayable width (in millimeters).

hf_mheight Displayable height (in millimeters).

hf_bperpel Bits per pel (1, 2, or 4).

hf_phdevid Display device ID.

Physical Display Font Information

Field **Value**

hf_numfont Number of fonts available to this display. The following fields appear for each available font.

hf_fontid Physical font ID.

hf_fontstyle Physical font style.

HFFNTVAR This font results in a variable presentation space depending on the display type used.
HFFNTKSR This font results in a 80x25 presentation space regardless of the display type used.

Field **Value**

hf_fontattr[3] Physical font attribute. This field may have the following values:

HFFNTPLAIN Plain
HFFNTBOLD Bold
HFFNTITALIC Italic.

hf_fontwidth Physical font width (the width of a character cell in pels).

hf_fontheight Physical font height (the height of a character cell in pels).

Physical Display Color Information

Field	Value
hf_numcolor	Total number of colors possible
hf_numactive	Number of colors that can be active at any one time
hf_numfgrnd	Number of foreground color options
hf_numbgrnd	Number of background color options
hf_actcolor	Active color value. The value of this field can be in the range 0 to the total number of colors possible (**hf_numcolor**) minus 1. This field is repeated for each of the currently active colors.

Query Locator Command:

To query the locator, use the **hfqgraphdev** structure with fields set as follows:

Field	Value
hf_intro.hf_typehi	**HFQLOCCH**
hf_intro.hf_typelo	**HFQLOCCL**

This command returns a **hfqlocr** structure with the following fields:

Field	Value
hf_intro.hf_typehi	**HFQLOCRH**
hf_intro.hf_typelo	**HFQLOCRL**
hf_resolution	The resolution of the locator (a 4-byte value) in millimeters per 100 count.
hf_devinfo[0]	Locator attributes: **HFLOCABS** If set, absolute coordinates (tablet). If not set, relative coordinates (mouse). **HFLOCUNKNOWN** Unknown sensor type, or the locator is a mouse. **HFLOCSTYLUS** The tablet has a stylus sensor. **HFLOCPUCK** The tablet has a puck sensor.
hf_horzmax_cnt	Horizontal maximum count (a 2-byte value).
hf_vertmax_cnt	Vertical maximum count (a 2-byte value).
hf_horzdead_zone	Horizontal tablet dead zone or mouse threshold.
hf_vertdead_zone	Vertical tablet dead zone or mouse threshold.

Query Presentation Space Command:

This data determines how to define a block of characters in the presentation space to query. Attribute and character set information on the queried block are returned. This query is valid only in KSR mode.

The **hfqpresc** structure is used for this command,

Field	**Value**
hf_intro.hf_typehi	**HFQPRESCH**
hf_intro.hf_typelo	**HFQPRESCL**
hf_sublen	2
hf_subtype	0
hf_xuleft	The upper-left X coordinate (first column of the block)
hf_yuleft	The upper-left Y coordinate (first row in the block)
hf_xlright	The lower-right X coordinate (last column in the block)
hf_ylright	The lower-right Y coordinate (last row in the block).

The data returned from this command is an ASCII data stream that contains character codes from the queried block. Character set and attribute changes are indicated with **SGR** and **SG0** control sequences. A line feed control is returned after the last character code in each line of the queried block. This command is useful when you want to save the attributes of a screen for restoring at some later time.

Note: The returned attributes may be only a subset of the original attributes specified for query. The subset in this case is those attributes actually supported by the physical device.

The response is returned in an **hfqpresr** structure, which contains the following fields:

Field	**Value**
hf_intro.hf_typehi	**HFQPRESRH**
hf_intro.hf_typelo	**HFQPRESRL**

The response contains an ASCII data stream that includes all ASCII data currently associated with the input buffer.

Query HFT Device Command:

This command gets information about the **hft** device. To issue this command, use the **hfqhftc** structure with fields set as follows:

Field	**Value**
hf_intro.hf_typehi	**HFQHFTCH**

hf_intro.hf_typelo	**HFQHFTCL**

The command returns an **hfqhftr** structure with the following fields:

Field	**Value**
hf_intro.hf_typehi	**HFQHFTRH**
hf_intro.hf_typelo	**HFQHFTRL**
hf_phdevid	Physical display device ID (the same as returned by "Query Device IDs Command")
hf_phrow	Number of character rows, based on the current font
hf_phcol	Number of character columns, based on the current font
hf_phcolor	Number of colors allowed on the display
hf_phfont	Number of fonts defined in the system
hf_phkbdid	Physical keyboard ID: **0** 101-key keyboard **1** 102-key keyboard

2.5.11.6 Screen Manager ioctls

Refer to the **hftqsm.c** and **hftcsm.c** programs for examples of the topics discussed in this section.

Subtopics

2.5.11.6.1 Query Screen Manager (HFQSMGR)

Queries the screen manager. The file descriptor must be associated with the screen manager, that is, **/dev/hft/mgr**. This is invoked by the following:

```
int ioctl(fildes, HFQSMGR, arg)
int fildes;
struct hfbuf *arg;

struct hfbuf
{
   char *hf_bufp;
   int hf_buflen;
};
```

The contents of the following **hfqstat** structure are stored in the memory area pointed to by **hf_bufp**.

```
struct hfqstat
{
   short hf_numvts;
   struct hfvtinfo
   {
     unsigned short hf_vtiodn;
     unsigned short hf_vtstate;
   } hf_vtinfo[HFNUMVTS];
};
```

Field	Description
hf_numvts	The number of virtual terminals.

The following fields are repeated for each virtual terminal:

hf_vtiodn	The virtual terminal IODN.
hf_vtstate	Status:
	HFVTHIDDEN The virtual terminal is hidden.
	HFVTACTIVE The virtual terminal is active.
	HFVTCOMMAND The virtual terminal is the command terminal.

2.5.11.6.2 Control Screen Manager (HFCSMGR)

This commands the screen manager which controls the status of virtual terminals. Virtual terminals are linked together in a group called the screen manager ring. The screen manager places an entry in the ring for each virtual terminal opened. The terminal that is currently active is called the head of the ring; the last terminal on the ring is called the tail. When a new terminal is added to the ring, that terminal becomes the head of the ring.

Three key sequences switch between virtual terminals and control which terminal is currently active. The active terminal is the terminal that accepts keyboard or locator input and updates the physical display. Pressing the **Alt + Action** keys on the active terminal makes the next virtual terminal active. This relationship is indicated by **a** in Figure 5-1. Pressing the **Shift + Action** keys on the active terminal makes the last virtual terminal active. The **b** in Figure 5-1 indicates this relationship. Pressing the **Ctrl** and **Action** keys on the active virtual terminal make the command virtual terminal active.

Figure 5-1. Screen Manager Ring Examples. In this figure, **a** indicates the path from the active virtual terminal to the next and **b** indicates the path from the active virtual terminal to the last.

Note that with three entries in the ring, all the terminals can be accessed with a single key sequence. With four or more entries, terminals can be skipped in some cases to activate a particular terminal. For example, in the preceding figure with four terminal entries, terminal number 2 cannot be accessed from the active terminal number 4 without first skipping to terminal number 1 or terminal number 3.

The hide option of this command logically removes terminals from the ring. Hiding a terminal causes it to be bypassed when its position in the ring would ordinarily make it the active terminal.

The file descriptor must be associated with a screen manager, that is, **/dev/hft/mgr**. This is invoked by the following:

```
int ioctl(fildes, HFCSMGR, arg)
int fildes;
struct hfsmgrcmd *arg;

struct hfsmgrcmd  {
    int hf_cmd;
    int hf_vtid;
    int hf_vsid;
};
```

The **hf_vtid** and **hf_vsid** fields are set as follows:

hf_vtid The IODN of the virtual terminal

hf_vsid 0.

The **hf_cmd** field contains one of the following screen manager commands:

SMACT Activates the virtual terminal. This command places the virtual terminal specified by the IODN at the head of the

screen manager ring, making it the active terminal. The terminal's hidden flag is also cleared. The screen manager cannot activate the virtual terminal if the currently active virtual terminal cannot be deactivated.

SMHIDE Hides the virtual terminal. This command marks the terminal identified by the IODN so that the screen manager will not activate it. This does not affect the terminal's position in the ring. When the hidden flag is set, the screen manager ignores the terminal's presence in the ring until an **SMUNHIDE** command is issued. If the virtual terminal is active when the hide command is issued, then the screen manager makes the terminal inactive. Hiding the active virtual terminal has the same effect as the **last** window function. If all virtual terminals are hidden, then the physical display continues to show the contents of the last virtual terminal that was hidden.

SMSCMD Sets the command virtual terminal. This command designates a terminal as the command virtual terminal. The command virtual terminal is the terminal that is activated by pressing both locator buttons at the same time, or by pressing the **Ctrl-Action** key sequence.

SMUNHIDE Undoes the action performed by **SMHIDE**. The **hf_vtid** field contains the IODN of the virtual terminal where the command should be sent. The **hf_vsid** field is reserved.

This command restores the presence of the terminal in the ring, but does not affect its ring position or make it active. If the virtual terminal happens to be at the head of the ring when this command is issued, then it becomes visible and active.

SMCVTEN Causes the command virtual terminal to be activated when both locator buttons are pressed at the same time. This is the default setting. Since all virtual terminals are affected, programs that change this setting should restore it as soon as the locator is no longer needed.

SMCVTDI Undoes the function done by **SMCVTEN**. Causes input data to be reported when both locator buttons are pressed at the same time. The data reported is similar to that reported when a single button is pressed.

2.5.11.7 Virtual Terminal Commands

Application programs control the behavior of their virtual terminals either through ioctls or by writing VTD character sequences into the output stream.

ASCII data can be sent to the virtual terminal using the **write** system call along with data of any length. In addition, virtual terminal control structures are sent to the virtual terminal using the **write** system call.

Subtopics
2.5.11.7.1 VTD Control Structure
2.5.11.7.2 Set KSR Color Palette
2.5.11.7.3 Change Fonts
2.5.11.7.4 Set Cursor Representation
2.5.11.7.5 Set Keyboard LEDs
2.5.11.7.6 Set Protocol Modes

2.5.11.7.1 VTD Control Structure

Each control structure is introduced by a virtual terminal data (VTD) character sequence. The VTD prefix consists of the ASCII codes ESC, [, and **x** (0x1B5B78). This prefix is followed by a length and an operation type code. The data that follows this structure depends on the type of control.

The **hfintro** structure looks like this:

```
{
    char hf_esc;
    char hf_lbr;
    char hf_ex;
    char hf_len[4];
    char hf_typehi;
    char hf_typelo;
};
```

The significant fields in the **hfintro** structure are:

hf_len The total number of bytes in the header and associated data, not including the three-character VTD control sequence. In other words, the length is the total number of characters in the control sequence minus 3.

hf_typehi The high-order byte of the information type code.

hf_typelo The low-order byte of the information type code.

The values of **hf_typehi** and **hf_typelo** are documented with each command.

Because the **hfintro** structure is an odd number of bytes in length, it is designated as the character array **hf_intro[HFINTROSZ]** in the structures that define the various operation requests. This prevents the C compiler from inserting bytes into the structure to align the following fields on word boundaries. The **hf_typehi** and **hf_typelo** fields are referred to **hf_intro.hf_typehi** and **hf_intro.hf_typelo** in this book, although these references are not precisely correct.

All reserved and unused fields must be set to 0. You can set the entire structure to 0 and then fill in the appropriate fields.

2.5.11.7.2 Set KSR Color Palette

This command specifies the color to associate with certain display adapters. The default color palettes are the ANSI 3.64 palette for character terminals and the PC color palette for all-points-addressable terminals. If the color specified is not supported by the adapter, the virtual display driver sets that color to the default for that mode.

The structure for this command is **hfcolorpal**, and it contains the following fields:

Field	Value
hf_intro.hf_typehi	**HFCOLORPALH**
hf_intro.hf_typelo	**HFCOLORPALL**
hf_sublen	2
hf_subtype	1
hf_numcolor	Number of entries in the palette
hf_palet	Adapter-specific settings of the first entry in the color palette. These settings must be repeated for each entry of the color palette corresponding to the display adapter. See the appropriate hardware technical reference for information about the display adapter.

The file **/usr/lib/samples/hft/ksrpal.c** contains a sample program that illustrates how to use this function with the VGA adapter.

2.5.11.7.3 Change Fonts

When a virtual terminal is first opened, and whenever it is changed, the default font is used. AIX PS/2 allows the user to change between two built-in 80 column by 25 row fonts. The default font (1025) uses the hardware of the VGA to produce the symbols in code page P0 (see "data stream" in topic 2.4.3). The other font (1026) uses the APA mode of the VGA adapter to produce all the symbols in code pages P0, P1, and P2.

Note that if the font is changed, the data currently in the presentation space is lost, and the cursor reverts to the double underscore and is placed at the home position (first column, first row). If it is desirable to control fonts, the fonts should be explicitly set when opening a terminal or changing a display.

Field	**Value**
hf_intro.hf_typehi	**HFFONTH**
hf_intro.hf_typelo	**HFFONTL**
hf_sublen	2
hf_subtype	1
hf_primary	Physical font ID of primary font attribute.
hf_alt1	Reserved.
hf_alt2	Reserved.
hf_alt3	Reserved.
hf_alt4	Reserved.
hf_alt5	Reserved.
Field	**Value**
hf_alt6	Reserved.
hf_alt7	Reserved.

Possible graphic renditions of VGA adapter:

	Hardware Font 1025	**Software Font 1026**
Bold or Bright	Yes	Yes
Underscore	No	Yes
Slow Blink	Yes	No
Reverse Image	Yes	Yes
Alternate Fonts	No	No
Code Page P0	Yes	Yes
Code Pages P1 and P2	No	Yes

2.5.11.7.4 Set Cursor Representation

The cursor representation data format determines how the cursor is presented on the display screen. The **hfcursor** structure is used for this request:

Field	Value
hf_intro.hf_typehi	**HFCURSORH**
hf_intro.hf_typelo	**HFCURSORL**
hf_sublen	2
hf_subtype	0
hf_rsvd	Reserved.
hf_shape	Cursor shape:

HFNONE	No cursor.
HFSINGLUS	Single underscore.
HFDBLUS	Double underscore.
HFHALFBLOB	Lower half of illuminated character cell.
HFMIDLINE	Double mid-character line.
HFFULLBLOB	Full illuminated character cell.

2.5.11.7.5 Set Keyboard LEDs

The structure for this command is **hfkled**, and it contains the following fields:

Field	Value
hf_intro.hf_typehi	**HFKLEDCH**
hf_intro.hf_typelo	**HFKLEDCL**
hf_sublen	2
hf_subtype	1
hf_ledselect	Indicates which of three LEDs to change: **HFNUMLOCK** The **Num Lock** LED **HFCAPSLOCK** The **Caps Lock** LED **HFSCROLLOCK** The **Scroll Lock** LED.

Field	Value
hf_ledvalue	Indicates the value to which to set the LEDs specified in **hf_ledselect**. LEDs that are specified with a 1 bit are set: **HFNUMLOCK** The **Num Lock** LED **HFCAPSLOCK** The **Caps Lock** LED **HFSCROLLOCK** The **Scroll Lock** LED.

The **hftskleds.c** sample program has an example of this command.

2.5.11.7.6 Set Protocol Modes

Refer to the **hftinput.c** sample program for an example of the topics discussed in this section.

Protocol mode settings determine how the virtual terminal will interpret coded data, translate and return input data. Two bits control each mode. The first, in the **hf_select** field, indicates whether to use the current mode setting. If this bit is set, then the corresponding bit in **hf_value** indicates the new setting for the mode. The mode bits are set to the default value when the virtual terminal is opened.

The **hfprotocol** structure gives the protocol definitions:

Field	**Value**
hf_intro.hf_typehi	**HFKSRPROH** or **HFMOMPROH**
hf_intro.hf_typelo	**HFKSRPROL** or **HFMOMPROL**
hf_sublen	2
hf_subtype	0
hf_select	Specifies which modes to change. A bit value of 1 specifies the mode represented by that bit to change.
hf_select[0]	Mode selectors: **HFHOSTS** **HFXLATKBD**
hf_select[1]	Mode selectors: **HFWRAP** **HFLOCATOR**
hf_value[0]	New mode values: **HFHOSTS** **HFXLATKBD**
hf_value[1]	New mode values: **HFWRAP** **HFLOCATOR**

When issuing this command, specify a type of either **HFKSRPROH**, **HFKSRPROL** or **HFMOMPROH**, **HFMOMPROL** depending on whether you are sending this command from within Keyboard Send-Receive mode (KSR) or Monitor mode (MOM). Only certain protocol modes are valid in each of these modes, as shown in the following table. An attempt to set an invalid protocol mode results in rejection of the entire request.

Protocol Mode	**When Valid**	**Meaning**
HFHOSTS	KSR	A 0 bit (default) means not to report shift key depressions. A 1 bit means report shift key

depressions.

HFHOSTS mode specifies whether to report keyboard status changes. If **HFHOSTS** mode is set, the keyboard status information is returned in the KSI ANSI control (see "Multi-Byte Controls" in topic 2.4.3.3.2).

HFXLATKBD	KSR, MOM	A 1 bit (default) specifies that the keyboard input is translated. A 0 bit indicates send key data as untranslated key controls.
HFWRAP	KSR	A 1 bit (default) causes the cursor to wrap when the presentation space boundary is exceeded. A 0 bit specifies do not wrap the cursor.
HFLOCATOR	KSR, MOM	A 0 bit (default) disables the locator from sending data. A 1 bit enables the locator to send data.

2.5.11.8 Configuring the Virtual Terminal

Some applications may require that the virtual terminal be set to other than the IBM-defined initial characteristics. The **hft** driver provides methods of querying the settings and capabilities of the virtual terminal, and methods of setting these to new values if required.

Earlier sections have described some of these methods:

- Query locator
- Set locator thresholds
- Query physical device (color, fonts, display attributes)
- Set KSR color palette
- Change fonts
- Set cursor representation
- Set keyboard LEDs
- Set protocol modes (keyboard translation, enable locator, etc.)
- Query and control Screen Manager

This section describes the initial default settings of each virtual terminal and facilities for changing the ones that have not already been described in prior sections:

- Initial State
- Reconfigure **ioctl**
  - Change keyboard typematic rate
  - Change keyboard typematic delay
  - Change locator sample rate
  - Replace keyboard position map
  - Replace Unique 1 and Unique 2 character set map
  - Replace echo/break map

If the physical display model is ever to be changed or if the physical keyboard (country keycap markings) are ever to be changed, then you should use the change parameters option of the AIX PS/2 boot to change them.

Subtopics

2.5.11.8.1 Initial State

When a new terminal is opened, it is initialized to a known default state. The initial terminal state is the following:

Mode: Keyboard Send-Receive (KSR)

Echo/Break Map: Echo all characters; break for none

Tab Rack: The first, every eighth, and the last position of ever line.

ASCII Controls

LNM	Set (line feed-new line mode)
IRM	Not set (insert mode)
SRM	Not set (send receive mode-set echo off)
TSM	Not set (tab stop mode)
CLM	Not set (carriage return-new line mode)
AUTONL	Set (wrap character to following line when end of correct line is reached)

Protocol

WRAP	Set (wrap cursor at boundary)
LOCATOR	Not set (do not return locator input)
XLATKBD	Set (translate keyboard input)
HOSTS	Not set (do not report keyboard status change)

Locator (Mouse) Threshold: 2.75 millimeters horizontal, 5. millimeters vertical.

Font: Provides a presentation space of 80 columns by 25 rows

Character mode color palette

Foreground		**Background**	
Entry	**Color**	**Entry**	**Color**
0	Black	0	Black
1	Red	1	Red
2	Green	2	Green
3	Yellow	3	Yellow
4	Blue	4	Blue
5	Magenta	5	Magenta
6	Cyan	6	Cyan
7	White	7	White
8	Gray		
9	Light red		
10	Light green		
11	Brown		
12	Light blue		
13	Light magenta		
14	Light cyan		
15	High intensity white		

2.5.11.8.2 Reconfigure (HFRCONF)

Refer to the notes in **/usr/lib/samples/README.conf** before using **Reconfigure**.

A user program can reconfigure the virtual terminal to include different real devices. This operation is invoked by the following:

```
int ioctl(fildes, HFRCONF, arg);
int fildes;
struct hfrconf *arg;

struct hfrconf
{
   unsigned hf_op;
   unsigned hf_obj;
   union
   {
     uint hf_infob;
     struct
     {
        ushort hf_iodn;
        ushort hf_iocn;
     } hf_info2;
   } hf_info;
   union
   {
     char *hf_cptr;
    struct hfbuf *hf_hfbufptr;
   } hf_mapinfo;
};
```

This command changes the configuration of the physical terminal or the virtual terminal defaults.

The **hf_op** field contains the requested operation. The valid operations appear in the following list. These reconfigure operations, with the exception of those followed by an * (asterisk), take effect only for terminals opened after the reconfiguration. The operations followed by an asterisk take effect for the terminals that are currently open as well as those opened after the reconfiguration.

HFCHGKBDRATE* Changes the keyboard typematic rate. Bits 24 - 31 of **hf_obj** indicate the keyboard typematic rate. For the standard PS/2 keyboard, valid values are between 2 and 30 characters per second and can be incremented in 1 character-per-second units. The default value for the PS/2 keyboard is 11 characters per second.

HFCHGKBDDEL* Changes the keyboard typematic delay. Bits 16 - 31 of **hf_obj** indicate the keyboard typematic delay. For the standard PS/2 keyboard, valid values are between 250 and 1000 milliseconds and can be incremented in 250 millisecond units. The default value for the PS/2 keyboard is 500 milliseconds.

HFCHGLOCRATE* Change locator sample rate. Bits 24 - 31 of **hf_obj** indicate the locator sample rate. For the standard PS/2 locator, valid values are 10, 20, 40, 60, 80, and 100 samples per second. The default for the PS/2 locator is 60

samples per second.

HFKEYMAP Replaces the keyboard position code map. **hf_mapinfo** contains the new position code map address. See the **/usr/lib/samples/hft/hftkbdmap.c** file.

HFDISPMAP Replaces the character code maps for the Unique 1 and Unique 2 character sets. **hf_mapinfo** contains the new unique character code map address. See the **/usr/lib/samples/hft/hftchrmap.c** file.

HFECHOMAP Replaces the echo/break map. **hf_mapinfo** contains the new echo/break map address. See the **/usr/lib/samples/hft/hftecbrmap.c** file.

2.5.11.8.3 Set User-Defined Character Set

The ASCII character set-to-display code (font) mapping of a virtual terminal can be altered. For each virtual terminal, the virtual terminal maintains character set mapping tables for two unique user-definable character sets called Unique One and Unique Two. These sets contain 256 ten-bit display symbol codes, and are activated by the SG0 or SG1 control (see "Multi-Byte Controls" in topic 2.4.3.3.2).

Note: Data is kept in display symbol form in the virtual terminal, and translation back to ASCII codes is done using the standard character set definitions, not Unique One or Two.

The **hfcharset** structure is used for character set definition, and it contains the following fields:

Field	**Value**
hf_intro.hf_typehi	**HFCHARSETH**
hf_intro.hf_typelo	**HFCHARSETL**
hf_sublen	2
hf_subtype	1
hf_setnum	User-defined character set **HFUNIQ1** Unique One (user-definable set 1) **HFUNIQ2** Unique Two (user-definable set 2)
hf_rsvd	Reserved
hf_code	10-bit display symbol code. This field may be repeated up to 256 times. See "display symbols" in topic 2.4.4 for the values of the display symbols.

2.5.11.8.4 Set Echo and Break Maps (HFSECHO)

Refer to the **hftsmap.c** sample program for an example of the topics discussed in this section.

HFSECHO sets the **hft** echo and break maps. Echoing displays the character associated with a keystroke on the screen or performs the function associated with a control. Breaking switches the input path from the monitor mode input ring buffer to the unsolicited ASCII datastream flow. Echoing applies only to KSR mode; breaking applies only to MOM mode. Echoing and breaking can be selectively enabled for each ASCII code point and multibyte control sequence. The default is to echo all characters and control sequences, but not to break on any of them.

The HFSECHO operation is invoked by the following **ioctl** call:

```
int ioctl(fildes, HFSECHO, arg)
int fildes;
struct hfbuf *arg;

struct hfbuf
{
    char *hf_bufp;
    int hf_buflen;
};
```

The **hf_bufp** field points to an array of 32 integers. The **hf_buflen** field contains the value 128 (0x80), which is the length of the array in bytes. The first sixteen integers constitute the echo map; the second sixteen integers are the break map.

Each of the two maps is treated as a set of bits. Bit 0 is the most significant bit of the first integer. Bit 511 is the least significant bit of the sixteenth integer. Each bit corresponds to an ASCII code point or multibyte control. Bits 0 through 255 (0xFF) correspond to the single-byte codes. Bits 256 (0x100) and higher correspond to multibyte control sequences, as illustrated in Figure 5-2. Bit 511 (0x1FF) specifies whether to echo or break on invalid and unsupported multibyte control sequences. See "data stream" in topic 2.4.3 for a detailed explanation of each of the multibyte control sequences.

Figure 5-2. Bit Positions of ASCII Controls in Echo Map

For the echo map, a bit set to 1 means the character or control sequence is echoed when a key that is mapped to it is pressed. The echo map is active only in KSR mode and can be set only from KSR mode.

For the break map, a bit set to 1 means that the character or control sequence is reported using the **read** system call instead of being placed in the input ring buffer. Also, the **SIGMSG** signal is sent to the process to indicate that input data is available. The break map is active only in monitor mode. (See "Monitor Mode (MOM)" in topic 2.5.11.4.2 for a description of the input ring buffer.)

The echo and break maps are shared by all code pages. For P0 graphic code points (0x20 to 0xFF), bits 32 to 255 (0x20 to 0xFF) of each map are used. For other code pages, each half of the code page is associated with bits 128 to 255 (0x80 to 0xFF). For example, bit 160 (0xA0) specifies the echo or break status of code points P0 0xA0, P1 0x20, P1 0xA0, P2 0x20, and P2

0xA0.

2.5.11.8.5 Set Keyboard Map (HFSKBD)

Refer to the **hftskch.c**, **hftskfn.c**, and **hftskst.c** sample programs for examples of the topics discussed in this section.

HFSKBD sets the keyboard map. Most keys on the keyboard can be remapped, changing the character or control sequence each key generates when pressed. See "keyboard" in topic 2.5.13 for additional details. This is invoked by the following:

```
int ioctl(fildes, HFSKBD, arg)
int fildes;
struct hfbuf *arg;

struct hfbuf
{
   char *hf_bufp;
   int   hf_buflen;
};
```

The **hf_bufp** field points to a **hfkeymap** structure, and **hf_buflen** contains its length.

```
struct hfkeymap {
        char  hf_rsvd1 ;
        char  hf_nkeys;
        struct hfkey {
                char hf_kpos;
                char hf_kstate;
                struct hfkeyasgn
                {
                     /* for single character */
                     char hf_pagenum;   /* Code page  */
                     char hf_character;  /* Character to map  */
#define hf_page      hf_pagenum
#define hf_char      hf_character
                     /* for function id */
#define hf_keyidh    hf_pagenum      /* high byte of id */
#define hf_keyidl    hf_character    /* low byte of id */
                     /* for character string */
#define hf_kstrl     hf_character    /* length of string */
                }hf_keyasn;
        } hfkey[HFNKEYS];
};
```

The **hfkeymap** structure can remap one or more keys, the number of which is specified by the **hf_nkeys** field. This many **hfkey** structures follow. **HFNKEYS**, which is used as the dimension for the **hfkey** array, is by default defined to be 1, allowing one key to be remapped. To change **HFNKEYS**, set its value in a **#define** statement that comes before the **#include <hft.h>** statement.

The **hfkey** structure contains information for each key being remapped, such as key position, shift states, and the type of remapping being done. The fields in the **hfkey** structure are:

hf_kpos The key position number. See "keyboard" in topic 2.5.13.

hf_kstate This field is subdivided into three groups of bits:

HFMAPMASK
Defines the bits that specify the type of mapping to be performed:

HFMAPCHAR	Specifies mapping a single character to a key.
HFMAPNONSP	Specifies mapping a nonspacing character to a key. (See "Nonspacing Characters" in topic 2.4.3.2 for information about nonspacing characters.)
HFMAPFUNC	Specifies mapping a function ID to a key.
HFMAPSTR	Specifies mapping a string of more than one character to a key. Maximum string size is 256 bytes.

HFSHFMASK
Defines the bits that specify the shift state that applies to the key being mapped:

HFSHFNONE	Specifies the base state (no shift state)
HFSHFSHFT	Specifies the shift state
HFSHFCTRL	Specifies the **Ctrl** state
HFSHFALT	Specifies the **Alt** state
HFSHFALTGR	Specifies the **Alt Gr** (Alternate Graphics) state.

HFCAPSL
Specifies whether the **Caps Lock** state affects the key. If set, then when **Caps Lock** mode is on, the base state of a key functions as the shift state, and the shift states functions as the base state.

The **hfkeyasgn** structure specifies the key to be remapped and the character codes generated when the key is pressed or released. The fields of this structure differ depending on the value of the **HFMAPMASK** bits in **hf_kstate**:

HFMAPCHAR, HFMAPNONSP:

hf_page	Specifies the code page < P0 Display symbols 32-255 = P1 Display symbols 256-479 > P2 Display symbols 480-703
hf_char	Specifies a character (also called a code point) in that code page.

HFMAPSTR:

hf_page	Specifies the code page < P0 Display symbols 32-255 = P1 Display symbols 256-479 > P2 Display symbols 480-703
hf_kstrl	Specifies (**the length of the string in bytes**) minus 1.

This is immediately followed by the string.

Note: Due to limitations of the **hfkeymap** structure, only one key can be assigned a string value, and it must be the last key specified in the **hfkey** array. This is because the structure itself does not contain space for the variable-length string,

but the string must immediately follow the structure in memory.

See "Mapping Multiple Strings" if you wish to set more than one string using **HFSKBD**. You can also make repeated **HFSKBD ioctl**s with different assignments.

HFMAPFUNC:

hf_keyidh	Specifies the high-order byte of the function ID.
hf_keyidl	Specifies the low-order byte of the function ID.

The following list gives the function IDs for each of the functions that can be assigned to keys. See "Multi-Byte Controls" in topic 2.4.3.3.2 for more details about these functions.

ID	**Name**
0x0000 -- 0x00FE	(PFK) Issues the Programmable Function Key sequence for PF key 1 (ID = 0x0000) through 255 (ID = 0x00FE).
0x0101	(CUU) Moves the application cursor up one line.
0x0102	(CUD) Moves the application cursor down one line.
0x0103	(CUF) Moves the application cursor forward one character.
0x0104	(CUB) Moves the application cursor backward one character.
0x0105	(CBT) Moves the application cursor to the previous horizontal tab stop or beginning of field.
0x0106	(CHT) Moves the application cursor to the next horizontal tab stop or beginning of field.
0x0107	(CVT) Moves the application cursor down one vertical tab stop.
0x0108	(HOME) Moves the application cursor to the first line, first character in the presentation space.
0x0109	(LL) Moves the application cursor to the last line, first character in the presentation space.
0x010A	(END) Moves the application cursor to the last line, last character in the presentation space.
0x010B	(CPL) Moves the application cursor to the first character of the previous line.
0x010C	(CNL) Moves the application cursor to the first character of the next line.
0x0151	(DCH) Deletes the character over the application cursor.
0x0152	(IL) Inserts one line following the line of the application cursor.
0x0153	(DL) Deletes the line of the application cursor.
0x0154	(EEOL) Erases to the end of the line.
0x0155	(EEOF) Erases to the next tab stop.
0x0156	(CLEAR) Erases all characters from the presentation space.
0x0157	(INIT) Restores the initial state of the virtual terminal. (See the description of RIS in "Multi-Byte Controls" in topic 2.4.3.3.2.)
0x0162	(RI) Performs one line reverse index control.
0x0163	(IND) Performs one line index control.
0x01FF	(IGNORE) Sends no information when the key is pressed.

Note: On the U.S. 101-key keyboard, the left **Alt** key produces the **Alt** shift state, and the right **Alt** key produces the **Alt Gr** shift state. The default keyboard mapping for the **Alt** and **Alt Gr** states is identical for all keys.

If a U.S. 101-key keyboard is attached, then mapping the **Alt** state of a

key automatically causes the same mapping to be assigned to the **Alt Gr** state. This allows the two **Alt** keys on the U.S. keyboard to function identically for most applications. If you want to remap both the **Alt** and **Alt Gr** states of a key, you must remap the **Alt** state first, then the **Alt Gr** state. Software written primarily for keyboards other than the U.S. keyboard should remap the states in this order to ensure compatibility.

Mapping Multiple Strings:

The **hft** device driver allows any number of keys to be assigned string values with **HFSKBD**. You do this by setting up your own key map buffer instead of using the **hfskeymap** structure.

The following example shows a data structure which can be used to map two keys to strings:

```
char map [] = {
   0x00,      /* Reserved */
   0x02,      /* Number of keys = 2 */
   /* Key pos, shift state, code page, string length, string */
   108, HFMAPSTR, '<', 3, '\033', 'O', 'M',
   105, HFMAPSTR, '<', 3, '\033', 'O', 'm'
};
```

This data structure can be used to assign key position 108 and 105 to string "Escape O M" and "Escape O m", respectively.

2.5.11.9 termio Support

Input modes described in "termio" in topic 2.5.28 supported are INLCR, IGNCR, ICRNL, IUCLC, IXON, and IXANY. Input modes IGNBRK and BRKINT are not supported because there is no **Break** key. Input modes IGNPAR, PARMRK, and INPCK are not supported because parity is not provided. Input mode ISTRIP is not supported either. ICRNL is supported by using the keyboard remap facility to change the code sent by the **Enter** (**Return**) and **Ctrl-M** keys. Also, the implementation of IXON is different. If the user presses **Ctrl-S** while output is being performed on the screen, the output does not stop until the end of the current **write** system call.

Output modes supported are OPOST, ONLCR, and OCRNL. The delay insertion, parity, and stop bit modes are not supported.

Line discipline modes supported are ISIG, ICANON, ECHO, ECHOE, ECHOK, ECHONL, NOFLSH, and Enhanced Edit Mode.

Screen paging is also supported using the TCGLEN and TCSLEN **ioctl** operations. When paging is active, the contents of the buffer supplied by the **write** call are written out in page-size pieces.

Other **ioctl** operations supported by **hft** include TCXONC and TCFLSH. The TCSBRK operation is not supported.

2.5.11.10 select Support

The **hft** device driver supports the **select** system call in the following manner:

Read selects are satisfied when input data is available

Write selects are always satisfied immediately

Exception selects are never satisfied, or hang indefinitely if n **timeout** value is specified.

See "select" in topic 1.2.242 for more information about this system call.

2.5.11.11 Considerations for hft Emulation

Refer to the sample program **hftctl.c** for examples of the topics discussed in this section.

Communicating with an emulated or remote **hft** device presents a unique situation because the **ioctl** system call cannot be used. This is a result of the fact that **ioctl** passes data directly to the virtual terminal subsystem, bypassing the data stream. An **hft** emulator is usually connected through a pseudo-TTY device, which means that all communication with it must be done through the data stream. Pseudo-TTY devices are discussed under "pty" in topic 2.5.21.

Therefore, two special multibyte control sequences can be used in place of invoking the **ioctl** system call, allowing applications to request an emulated **hft** to perform the **ioctl** operations. However, the **hft** device driver, which controls the local console, does not recognize these control sequences; you must still use **ioctl** to perform these operations on an **hft** device that is not emulated.

Both of these multibyte control sequences begin with a virtual terminal data (VTD) header. VTDs are explained under "Virtual Terminal Commands" in topic 2.5.11.7.

To perform an **hft ioctl** operation whether or not the **hft** is emulated, an application should do the following:

1. Determine whether the **hft** device is being emulated. If the call **ioctl (fildes, IOCTYPE, 0)** returns the value **DD_PSEU**, then **fildes** is a pseudo-tty device, which means that **fildes** may be connected to an **hft** emulator. Otherwise, the **hft** device is not emulated.

2. If the **hft** is not emulated, then issue a regular **ioctl** system call, as outlined in "ioctl Operations" in topic 2.5.11.5.

3. If the **hft** is emulated, then do the following:

   a. Set the pseudo-tty for raw data. That is, disable all input and output processing. This is necessary because the control sequences can contain binary data that would be misinterpreted by the pseudo-tty device driver as ASCII control codes. See "termio" in topic 2.5.28 for details.

   b. Use the **write** system call to send an **hfctlreq** VTD structure, immediately followed by the request structure, if any, that would normally be pointed to by the **ioctl arg** parameter. The **hfctlreq** structure contains the following fields:

Field	**Value**
hf_intro.hf_typehi	**HFCTLREQH**
hf_intro.hf_typelo	**HFCTLREQL**

Field	**Value**
hf_request	The request type.
hf_arg_len	The length of the argument structure that follows the **hfctlreq** VTD, or 0 if none.

hf_rsp_len	The maximum length of the response data structure that is to be returned with the **hfctlack** VTD. This value is 0 if no response buffer is expected.

c. Read (using the **read** system call) until an acknowledgement VTD is received. This acknowledgement takes the form of an **hfctlack** structure, which is sometimes followed by a returned data structure, depending on the operation requested. The **hfctlack** structure contains the following fields:

Field	**Value**
hf_intro.hf_typehi	**HFCTLACKH**
hf_intro.hf_typelo	**HFCTLACKL**
hf_request	The type of request that is being acknowledged.
hf_ret_code	The error code: 0 indicates successful completion; a nonzero value is the value that is normally found in **errno**.
hf_arg_len	The length of the response data structure that follows the **hfctlack** VTD, or 0 if none. The length must not exceed the value of **hf_rsp_len** that was specified in the **hfctlreq** structure.

2.5.11.12 AIX PS/2 HFT Compatibility with AIX RT

In general, **hft** for AIX PS/2 is compatible with **hft** for AIX RT. However, there are a few areas of difference which are due to differences in the hardware of these systems, or due to the absence of a VRM Operating System layer on the PS/2. Highlights of the differences are as follows:

Differences Due to Hardware:

There is no support for Lighted Program Function Keys (LPFKs), dials or tablets.

The keyboard click is not settable

The sound volume is not settable

Only the VGA adapter of the PS/2 is supported. It has characteristic similar to the EGA adapter on the RT, however the PS/2 implementation allows a choice of two built-in fonts, one utilizing hardware character generation and the other using software character formation. The hardware font is faster and more efficient than the software font, but is limited to code page P0. The software font is slower and uses more CPU cycles, but is capable of generating all 616 symbols in code pages P0, P1, and P2.

There is no **/dev/bus** pseudo-device. Applications that use Monitor Mode will get the virtual address of the display adapter memory returned in the structure used to enter Monitor Mode (the HFSMON **ioctl**).

Differences Due to VRM:

There is limited reconfigurability of physical devices and drivers. It is not possible to replace individual virtual terminal subsystem components. Relationships and allocation of **hft** channels and IODNs are not the same. Applications should not make any assumptions about this or about the ordering of these numbers. For compatibility, applications should use **open** and accept the virtual terminal that is allocated.

Subtopics

2.5.11.12.1 Compatibility Table

The following table shows the compatibility between AIX PS/2 and AIX RT for each **hft** operation. Refer to the list of explanatory "Notes" which follows the table.

HFT Operation	**RT Compatible?**	**Notes**
HFQEIO Query I/O Error	Yes	
HFQDEV Query Device	No	1,2
HFRCONF Reconfigure		
HFADDLOC	Not Implemented	1,4
HFADDSOUND	Not Implemented	1,4
HFADDDISPLAY	Not Implemented	1,4
HFDELDISPLAY	Not Implemented	1,4
HFADDFONT	Not Implemented	1,4
HFCHGKBDRATE	Yes	
HFCHGKBDDEL	Yes	
HFCHGLOCRATE	Yes	
HFCHGCLICK	Yes	6
HFCHGVOLUME	Yes	6
HFKEYMAP	No	8
HFDISPMAP	No	8
HFECHOMAP	No	8
HFDEFAULT	Not Implemented	1,4
HFSETDD	Not Implemented	1,4
HFADDDIALS	Not Implemented	1,3
HFADDLPFK	Not Implemented	1,3
HFCHNGDMA	Not Implemented	1,3
HFGCHAN Get Channel Number	Yes	
HFSECHO Set Echo/Break Maps	Yes	
HFSKBD Set Keyboard Map	Yes	
HFGETID Get Virtual Term ID	Yes	
HFQUERY Query Command		
HFQDEVID	Yes	
HFQPDEV	Yes	
HFQLOC	Yes	
HFQLPFK	Yes	9
HFQDIALS	Yes	9
HFQPRES	Yes	
HFQHFT	Yes	
HFQDMA	Not Implemented	1,3
HFESOUND	Yes	
HFDSOUND	Yes	
HFSMON	Yes	7
HFCMON	Yes	
HFQSMGR	Yes	

HFCSMGR		Yes	
HFMDMA		Not Implemented	1,3
HFKLED	Set Keyboard LEDs	Yes	
HFLOTH	Set Loc Threshold	Yes	
HFSOUND	Sound	Yes	
HFCANSND	Cancel Sound	Yes	
HFCHGDSP	Change Phys Disp	Yes	6
HFKSRPRO	KSR Protocol Modes	Yes	
HFMOMPRO	MOM Protocol Modes	Yes	
HFTDZ	Set Tablet Dead Zones	Yes	9
HFLPFKS	Enab/Disab LPFK's	Yes	9
HFDIALS	Set Dial Granularity	Yes	9
HFCHARSET	Uniq1/Uniq 2	Yes	
HFCOLORPAL	Set KSR Color Palette	Yes	
HFFONT	Change Fonts	Yes	
HFCURSOR	Cursor Representation	Yes	
HFMOMREQ	Scrn Req Input Ring	Yes	
HFMOMREL	Screen Release	Yes	

Notes:

1. Produces a compile time error if specified.

2. Not implemented due to dependence on VRM.

3. No PS/2 Hardware to support this function.

4. Not implemented in Release 1 of AIX PS/2.

5. Implemented in a different way due to dependence on VRM.

6. Capability does not exist; a null success status is returned and no action is taken.

7. Function now returns an address in the response structure.

8. User programs will need to be modified to use an address instead of an IOCN due to dependence on the VRM.

9. No PS/2 hardware for these functions, however the correct values will be returned to reflect lack of hardware.

2.5.11.12.2 Byte-Ordering Considerations

There are several instances in the **hft** driver where a binary integer is being passed in a string of characters. The integer will always be arranged such that the most significant byte comes first and the least significant byte comes last, regardless of the machine that is being used to execute the code.

You should not use a union to align an integer with a byte field. This technique is machine-dependent and is not portable between RT and PS/2.

Consider an integer containing the hex value 0x12345678 stored at memory location 8. The stored memory image of this integer differs between RT and PS/2:

Figure 5-3. Stored memory differences between RT and PS/2

This difference in byte-ordering can get you in trouble. Assume that the character field **x** is known to contain an integer.

The following code should not be used:

```
union
{
   char c[4];
   int  i;
} u;
char  x[4];

u.c[0] = x[0];
u.c[1] = x[1];
u.c[2] = x[2];
u.c[3] = x[3];
```

What does **u.i** contain? On the RT, it contains the integer **x** and can safely be used. On the PS/2 it *does not* contain the integer **x** in the proper order. Bytes 0 and 3 are swapped and bytes 1 and 2 are swapped.

To get the correct results on both RT and PS/2, you should use code similar to the following:

```
int   i;
char  x[4];

i = ((x[0] << 24) + (x[1] << 16) + (x[2] << 8) + (x[3]));
```

This technique pulls the bytes from the character field in the proper order and uses the machine-independent shift instruction to position them in the integer. It will work properly for both types of machine. The same technique can be applied to retrieving **char[2]** and **char[3]** fields.

2.5.11.12.3 Sample Programs from AIX RT hft

Some of these sample programs did contain unions of integers and byte fields. They will not execute properly on the PS/2 and have been modified to follow the guidelines in Byte-Ordering Considerations. User programs based on these samples should be checked to see if such unions are being employed.

2.5.11.12.4 DOS Merge

Programs operating under control of DOS Merge participate in the HFT screen manager ring for hot-keying but otherwise cannot be controlled by **HFT ioctl**s or VTD sequences.

File
/dev/hft/*

Related Information
In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "fonts" in topic 2.3.19, "data stream" in topic 2.4.3, and "termio" in topic 2.5.28.

2.5.12 ilans

Purpose
Supports the ILANS network device driver.

```
# include <sys/devinfo.h>
# include <sys/b370/ilans.h>
# include <sys/b370/ilansceti.h>
```

Description
The **ilans** device driver is used to drive to the 9370 Integrated Token Ring Adatper. It is unique to AIX/370.

Since the supported device is a LAN network interface, normal network traffic is transmitted and received using the socket interface and not the device interface. The device interface is used only to monitor and control the operation of the device, via ioctl commands.

The device used to perform this monitor and control is **/dev/ilans#**, where # is the device minor number.

There is a small number of **ioctl** operations available besides the standard IOCTYPE and IOCINFO commands. They are:

DIOCONLINE
Bring the device on-line.

```
ioctl(fd, DIOCONLINE, arg)
char *arg
```

The argument is ignored.

DIOCOFFLINE
Take the device off-line.

```
ioctl(fd, DIOCOFFLINE, arg)
char *arg
```

The argument is ignored.

CIOGETSTATS
Return device statistics.

```
ioctl(fd, CIOGETSTATS, arg)
struct tkr_lstat *arg;
```

where **struct tkr_lstat** is declared as:

```
struct tkr_lstats {
 unsigned frames_tx;              /* frames transmitted */
 unsigned frames_rx;              /* frames received */
 unsigned mac_frames_tx;          /* mac frams transmitted */
 unsigned mac_frames_rx;          /* mac frams received */
 unsigned ri_frames_tx;           /* ri frames transmitted */
 unsigned ri_frames_rx;           /* ri frames received */
 unsigned line_error;             /* line error count */
 unsigned internal_error;         /* internal error count */
 unsigned burst_error;            /* burst error count */
 unsigned ari_fci_error;          /* ari fci count */
 unsigned abort_delimiter;        /* abort delimiter count */
```

```
    unsigned lost_frame;             /* count of lost frames */
    unsigned rx_congestion;          /* receive congestion count */
    unsigned frame_copied_error;     /* frame copy errors */
    unsigned frequency_error;        /* count of frequency error */
    unsigned token_error;            /* count of token errors */
    unsigned dma_bus_error;          /* count of dma bus errors */
    unsigned dma_parity_error;       /* count of dma parity errors *
    unsigned addr_unrecognized;      /* count of unrecognized addr *
    unsigned frame_not_copied_error; /* count of not copied */
    unsigned tx_strip_error;         /* count xmit strip errors */
    unsigned unauthorized_ap;        /* count of unauthorized ap */
    unsigned unauthorized_mf;        /* count of unauthorized mf */
    unsigned soft_error;             /* count of software errors */
    unsigned transmit_beacon;        /* count of transmit beacon */
    unsigned ioa_status_overrun;     /* count ioa status overrun */
    unsigned frames_discarded;       /* count of discarded frames */
    unsigned spurious_interrupts;    /* count spurious interrupt */
};
```

IEREPCTL

Control the VM EREP logging.

```
ioctl(fd, IEREPCTL, arg)
struct ilans_erep_ctl *arg;
```

where **struct ilans_erep_ctl** is declared as:

```
struct ilans_erep_ctl {
  unsigned char obr_thresh;
  unsigned char mdr_thresh;
  unsigned char obr_cnt;
     /* current log control count returned here */
  unsigned char mdr_cnt;
     /* current log control count returned here */
};
```

If a threshold is 0, no EREP logging is done. If set to 0xff, no change to the current state is made. Otherwise, logging set to occur every **n**th occurrence, where **n** is the number specified.

Related Information

In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137 and "open, openx, creat" in topic 1.2.199.

2.5.13 keyboard

Purpose
Maps the 101-Key PS/2 keyboard.

Description
A keyboard mapping table is maintained for each virtual terminal. This table relates a key indicated by its key position along with the shift, control, or alternate keys to a character, mode processor function or string of characters. Portions or all of this mapping table can be modified by data passed to the **hfbuf** structure in the HFSKBD type **ioctl** system call. See "hft" in topic 2.5.11 for information about this **ioctl** system call. See *AIX PS/2 Keyboard Description and Character Reference* for details about other PS/2 keyboards.

Each key on the standard PS/2 keyboard has a numeric position code that is used for this field. Figure 5-4 matches the key to its position code.

Figure 5-4. Position Codes for Remapping a Keyboard

The following keys are not redefinable:

Key Position	Function	States that cannot be remapped
30	Caps Lock key	All states
44	Left Shift key	All states
57	Right Shift key	All states
58	Control key	All states
60	Left Alternate key	All states
62	Right Alternate key	All states
64	Action key	Shift, Control, Alternate, and Alternate Graphics states
90	Num Lock key	Base and Shift states

Subtopics
2.5.13.1 US 101-Key Keyboard Translate Table
2.5.13.2 Keystroke Control Sequences for System Functions

2.5.13.1 US 101-Key Keyboard Translate Table

The following table gives this information about the default mapping for the U.S. 101-key keyboard:

Key Posn - Keyboard key position.

Shift - The shift state of the position: Base, Shift, Ctrl, or Alt. (Note that the Alt Gr shift state is always the same as the Alt state.)

Assignment - The character or control assigned to that key.

Keyboard Definition - Provides information as it would appear as part of a keyboard definition structure. (See "Set Keyboard Map (HFSKBD)" in topic 2.5.11.8.5 for details.) Within the table, interpret the fields as follows:

nnn - One-byte key position number being defined.

s - Shift state being defined:

b (base) - No **Shift** key is pressed.

s (shift) - Either left or right **Shift** key is pressed.

c (control) - **Ctrl** key is pressed.

a (alt) - **Alt** key is pressed.

t - Type of definition:

c - Regular character definition, followed by a 1-byte code page identifier and a 1-byte code point specification. The code page identifiers are:

< for Code Page P0
= for Code Page P1
> for Code Page P2

This identifier is followed by a 1-byte code point identifier, given in the table as a decimal number.

f - Function specification, followed by a 2-byte function identifier, which is specified in the table as a hexadecimal value.

s - String specification, followed by a 1-byte code page identifier, a 1-byte string length and the 1-byte code point identifiers within the specified code page. No string specifications are included in the default keyboard layouts.

d - Nonspacing or **dead** character definition, followed by a 1-byte code page identifier and a 1-byte code point specification. The code page identifiers are as described for character definition. This is followed by a 1-byte code point identifier, given in the table as a decimal number. There are no dead keys defined for the U.S. 101-key keyboard.

Returned String - Specifies the data that is returned to the

program that is reading the keyboard.

Notes - Specifies additional information. Entries in this column include:

CL - This key is affected by **CAPS LOCK**.

DK - This key is a dead character on this key state.

P1 - This key is a character from Code Page P1.

P2 - This key is a character from Code Page P2.

The **Alt** key, followed by one or more numbered keys on the numeric pad, will return a single character which has the value entered on the numeric pad. The value accumulates while the Alt key is held down and returns when that key is released. Only spacing character codes and single-byte controls are produced by this method.

Key Posn	Shift State	Assignment		Keyboard Definition nnn	s	t		I
1	Base	`	Grave Accent	1	b	c	< 96	0
1	Shift	~	Tilde Accent	1	s	c	< 126	0
1	Ctrl	PFK	57	1	c	f	0x39	E
1	Alt	PFK	115	1	a	f	0x73	E
2	Base	1	One	2	b	c	< 49	0
2	Shift	!	Exclamation Point	2	s	c	< 33	0
2	Ctrl	PFK	49	2	c	f	0x31	E
2	Alt	PFK	58	2	a	f	0x3a	E
3	Base	2	Two	3	b	c	< 50	0
3	Shift	@	At Sign	3	s	c	< 64	0
3	Ctrl	NUL	Null	3	c	c	< 0	0
3	Alt	PFK	59	3	a	f	0x3b	E
4	Base	3	Three	4	b	c	< 51	0
4	Shift	#	Number Sign	4	s	c	< 35	0
4	Ctrl	PFK	50	4	c	f	0x32	E
4	Alt	PFK	60	4	a	f	0x3c	E
5	Base	4	Four	5	b	c	< 52	0
5	Shift	$	Dollar Sign	5	s	c	< 36	0
5	Ctrl	PFK	51	5	c	f	0x33	E
5	Alt	PFK	61	5	a	f	0x3d	E
6	Base	5	Five	6	b	c	< 53	0
6	Shift	%	Percent Sign	6	s	c	< 37	0
6	Ctrl	PFK	52	6	c	f	0x34	E
6	Alt	PFK	62	6	a	f	0x3e	E
7	Base	6	Six	7	b	c	< 54	0
7	Shift	^	Circumflex Accent	7	s	c	< 94	0
7	Ctrl	SS2	Single Shift 2	7	c	c	< 30	0
7	Alt	PFK	63	7	a	f	0x3f	E
8	Base	7	Seven	8	b	c	< 55	0
8	Shift	&	Ampersand	8	s	c	< 38	0
8	Ctrl	PFK	53	8	c	f	0x35	E

8	Alt	PFK	64	8	a	f	0x40	E
9	Base	8	Eight	9	b	c	< 56	0
9	Shift	*	Asterisk	9	s	c	< 42	0
9	Ctrl	PFK	54	9	c	f	0x36	E
9	Alt	PFK	65	9	a	f	0x41	E
10	Base	9	Nine	10	b	c	< 57	0
10	Shift	(	Left Parenthesis	10	s	c	< 40	0
10	Ctrl	PFK	55	10	c	f	0x37	E
10	Alt	PFK	66	10	a	f	0x42	E
11	Base	0	Zero	11	b	c	< 48	0
11	Shift	)	Right Parenthesis	11	s	c	< 41	0
11	Ctrl	PFK	56	11	c	f	0x38	E
11	Alt	PFK	67	11	a	f	0x43	E
12	Base	-	Hyphen	12	b	c	< 45	0
12	Shift	_	Underscore	12	s	c	< 95	0
12	Ctrl	SS1	Single Shift 1	12	c	c	< 31	0
12	Alt	PFK	68	12	a	f	0x44	E
13	Base	=	Equal Sign	13	b	c	< 61	0
13	Shift	+	Plus Sign	13	s	c	< 43	0
13	Ctrl	PFK	69	13	c	f	0x45	E
13	Alt	PFK	70	13	a	f	0x46	E
14	Not available on keyboard							
15	Base	BS	Back Space	15	b	c	< 8	0
15	Shift	BS	Back Space	15	s	c	< 8	0
15	Ctrl	DEL	Delete	15	c	c	< 127	0
15	Alt	PFK	71	15	a	f	0x47	E
16	Base	HT	Horizontal Tab	16	b	c	< 9	0
16	Shift	CBT	Cursor Back Tab	16	s	f	0x105	E
16	Ctrl	PFK	72	16	c	f	0x48	E
16	Alt	PFK	73	16	a	f	0x49	E
17	Base	q	Lowercase q	17	b	c	< 113	0
17	Shift	Q	Uppercase q	17	s	c	< 81	0
17	Ctrl	DC1	Device Control 1	17	c	c	< 17	0
17	Alt	PFK	74	17	a	f	0x4a	E
18	Base	w	Lowercase w	18	b	c	< 119	0
18	Shift	W	Uppercase w	18	s	c	< 87	0
18	Ctrl	ETB	End Trans Block	18	c	c	< 23	0
18	Alt	PFK	75	18	a	f	0x4b	E
19	Base	e	Lowercase e	19	b	c	< 101	0
19	Shift	E	Uppercase e	19	s	c	< 69	0
19	Ctrl	ENQ	Enquiry	19	c	c	< 5	0
19	Alt	PFK	76	19	a	f	0x4c	E
20	Base	r	Lowercase r	20	b	c	< 114	0
20	Shift	R	Uppercase r	20	s	c	< 82	0
20	Ctrl	DC2	Device Control 2	20	c	c	< 18	0
20	Alt	PFK	77	20	a	f	0x4d	E
21	Base	t	Lowercase t	21	b	c	< 116	0

21	Shift	T	Uppercase t	21	s	c	< 84	0
21	Ctrl	DC4	Device Control 4	21	c	c	< 20	0
21	Alt	PFK	78	21	a	f	0x4e	E
22	Base	y	Lowercase y	22	b	c	< 121	0
22	Shift	Y	Uppercase y	22	s	c	< 89	0
22	Ctrl	EM	End of Media	22	c	c	< 25	0
22	Alt	PFK	79	22	a	f	0x4f	E
23	Base	u	Lowercase u	23	b	c	< 117	0
23	Shift	U	Uppercase u	23	s	c	< 85	0
23	Ctrl	NAK	Not Acknowledge	23	c	c	< 21	0
23	Alt	PFK	80	23	a	f	0x50	E
24	Base	i	Lowercase i	24	b	c	< 105	0
24	Shift	I	Uppercase i	24	s	c	< 73	0
24	Ctrl	HT	Horizontal Tab	24	c	c	< 9	0
24	Alt	PFK	81	24	a	f	0x51	E
25	Base	o	Lowercase o	25	b	c	< 111	0
25	Shift	O	Uppercase o	25	s	c	< 79	0
25	Ctrl	SI	Shift In	25	c	c	< 15	0
25	Alt	PFK	82	25	a	f	0x52	E
26	Base	p	Lowercase p	26	b	c	< 112	0
26	Shift	P	Uppercase p	26	s	c	< 80	0
26	Ctrl	DLE	Data Link Enabl	26	c	c	< 16	0
26	Alt	PFK	83	26	a	f	0x53	E
27	Base	[	Left Bracket	27	b	c	< 91	0
27	Shift	{	Left Brace	27	s	c	< 123	0
27	Ctrl	ESC	Escape	27	c	c	< 27	0
27	Alt	PFK	84	27	a	f	0x54	E
28	Base	]	Right Bracket	28	b	c	< 93	0
28	Shift	}	Right Brace	28	s	c	< 125	0
28	Ctrl	SS3	Single Shift 3	28	c	c	< 29	0
28	Alt	PFK	85	28	a	f	0x55	E
29	Base	\	Reverse Slash	29	b	c	< 92	0
29	Shift	|	Pipe Symbol	29	s	c	< 124	0
29	Ctrl	SS4	Single Shift 4	29	c	c	< 28	0
29	Alt	PFK	86	29	a	f	0x56	E
30	Base	Caps Lock		None				N
30	Shift	Caps Lock		None				N
30	Ctrl	Caps Lock		None				N
30	Alt	Caps Lock		None				N
31	Base	a	Lowercase a	31	b	c	< 97	0
31	Shift	A	Uppercase a	31	s	c	< 65	0
31	Ctrl	SOH	Start of Header	31	c	c	< 1	0
31	Alt	PFK	87	31	a	f	0x57	E
32	Base	s	Lowercase s	32	b	c	< 115	0
32	Shift	S	Uppercase s	32	s	c	< 83	0
32	Ctrl	DC3	Device Control 3	32	c	c	< 19	0
32	Alt	PFK	88	32	a	f	0x58	E
33	Base	d	Lowercase d	33	b	c	< 100	0

33	Shift	D	Uppercase d	33	s	c	< 68	0
33	Ctrl	EOT	End of Transmission	33	c	c	< 4	0
33	Alt	PFK	89	33	a	f	0x59	E
34	Base	f	Lowercase f	34	b	c	< 102	0
34	Shift	F	Uppercase f	34	s	c	< 70	0
34	Ctrl	ACK	Acknowledge	34	c	c	< 6	0
34	Alt	PFK	90	34	a	f	0x5a	E
35	Base	g	Lowercase g	35	b	c	< 103	0
35	Shift	G	Uppercase g	35	s	c	< 71	0
35	Ctrl	BEL	Bell	35	c	c	< 7	0
35	Alt	PFK	91	35	a	f	0x5b	E
36	Base	h	Lowercase h	36	b	c	< 104	0
36	Shift	H	Uppercase h	36	s	c	< 72	0
36	Ctrl	BS	Backspace	36	c	c	< 8	0
36	Alt	PFK	92	36	a	f	0x5c	E
37	Base	j	Lowercase j	37	b	c	< 106	0
37	Shift	J	Uppercase j	37	s	c	< 74	0
37	Ctrl	LF	Line Feed	37	c	c	< 10	0
37	Alt	PFK	93	37	a	f	0x5d	E
38	Base	k	Lowercase k	38	b	c	< 107	0
38	Shift	K	Uppercase k	38	s	c	< 75	0
38	Ctrl	VT	Vertical Tab	38	c	c	< 11	0
38	Alt	PFK	94	38	a	f	0x5e	E
39	Base	l	Lowercase l	39	b	c	< 108	0
39	Shift	L	Uppercase l	39	s	c	< 76	0
39	Ctrl	FF	Form Feed	39	c	c	< 12	0
39	Alt	PFK	95	39	a	f	0x5f	E
40	Base	;	Semicolon	40	b	c	< 59	0
40	Shift	:	Colon	40	s	c	< 58	0
40	Ctrl	PFK	96	40	c	f	0x60	E
40	Alt	PFK	97	40	a	f	0x61	E
41	Base	'	Quote, Apostrophe	41	b	c	< 39	0
41	Shift	"	Double Quote	41	s	c	< 34	0
41	Ctrl	PFK	98	41	c	f	0x62	E
41	Alt	PFK	99	41	a	f	0x63	E
42	Not available on keyboard							
43	Base	CR	Carriage Return	43	b	c	< 13	0
43	Shift	CR	Carriage Return	43	s	c	< 13	0
43	Ctrl	CR	Carriage Return	43	c	c	< 13	0
43	Alt	PFK	100	43	a	f	0x64	E
44	Base	Shift (Left)		None				N
44	Shift	Shift (Left)		None				N
44	Ctrl	Shift (Left)		None				N
44	Alt	Shift (Left)		None				F
45	Not available on keyboard							
46	Base	z	Lowercase z	46	b	c	< 122	0
46	Shift	Z	Uppercase z	46	s	c	< 90	0

46	Ctrl	SUB	Substitute Char.	46	c	c	< 26	0
46	Alt	PFK	101	46	a	f	0x65	E
47	Base	x	Lowercase x	47	b	c	< 120	0
47	Shift	X	Uppercase x	47	s	c	< 88	0
47	Ctrl	CAN	Cancel	47	c	c	< 24	0
47	Alt	PFK	102	47	a	f	0x66	E
48	Base	c	Lowercase c	48	b	c	< 99	0
48	Shift	C	Uppercase c	48	s	c	< 67	0
48	Ctrl	ETX	End of Text	48	c	c	< 3	0
48	Alt	PFK	103	48	a	f	0x67	E
49	Base	v	Lowercase v	49	b	c	< 118	0
49	Shift	V	Uppercase v	49	s	c	< 86	0
49	Ctrl	SYN	Synch Idle	49	c	c	< 22	0
49	Alt	PFK	104	49	a	f	0x68	E
50	Base	b	Lowercase b	50	b	c	< 98	0
50	Shift	B	Uppercase b	50	s	c	< 66	0
50	Ctrl	STX	Start of Text	50	c	c	< 2	0
50	Alt	PFK	105	50	a	f	0x69	E
51	Base	n	Lowercase n	51	b	c	< 110	0
51	Shift	N	Uppercase n	51	s	c	< 78	0
51	Ctrl	SO	Shift Out	51	c	c	< 14	0
51	Alt	PFK	106	51	a	f	0x65	E
52	Base	m	Lowercase m	52	b	c	< 109	0
52	Shift	M	Uppercase m	52	s	c	< 77	0
52	Ctrl	CR	Carriage Return	52	c	c	< 13	0
52	Alt	PFK	107	52	a	f	0x66	E
53	Base	,	Comma	53	b	c	< 44	0
53	Shift	<	Less Than Sign	53	s	c	< 60	0
53	Ctrl	PFK	108	53	c	f	0x6c	E
53	Alt	PFK	109	53	a	f	0x6d	E
54	Base	.	Period	54	b	c	< 46	0
54	Shift	>	Greater Than Sign	54	s	c	< 62	0
54	Ctrl	PFK	110	54	c	f	0x6e	E
54	Alt	PFK	111	54	a	f	0x6f	E
55	Base	/	Slash	55	b	c	< 47	0
55	Shift	?	Question Mark	55	s	c	< 63	0
55	Ctrl	PFK	112	55	c	f	0x70	E
55	Alt	PFK	113	55	a	f	0x71	E
56	Not available on keyboard							
57	Base	Shift (Right)		None				N
57	Shift	Shift (Right)		None				N
57	Ctrl	Shift (Right)		None				N
57	Alt	Shift (Right)		None				F
58	Base	Control		None				N
58	Shift	Control		None				N
58	Ctrl	Control		None				N
58	Alt	Control		None				N

```
59   Not available on keyboard

60   Base     Alternate Shift                None                          N
60   Shift    Alternate Shift                None                          N
60   Ctrl     Alternate Shift                None                          N
60   Alt      Alternate Shift                None                          N

61   Base     SP       Space                 61   b    c    < 32          0
61   Shift    SP       Space                 61   s    c    < 32          0
61   Ctrl     SP       Space                 61   c    c    < 32          0
61   Alt      SP       Space                 61   a    c    < 32          0

62   Base     Alternate Graphic Shift        None                          N
62   Shift    Alternate Graphic Shift        None                          N
62   Ctrl     Alternate Graphic Shift        None                          N
62   Alt      Alternate Graphic Shift        None                          N

63   Not available on keyboard

64   Base     PFK      114                   64   b    f    0x72          E
64   Shift    Previous Window                None                          P
64   Ctrl     Windows Window                 None                          W
64   Alt      Next Window                    None                          N

65 - Not available on keyboard

75   Base     PFK      139 INS Toggle        75   b    f    0x8b          E
75   Shift    PFK      139 INS Toggle        75   s    f    0x8b          E
75   Ctrl     PFK      140                   75   c    f    0x8c          E
75   Alt      PFK      141                   75   a    f    0x8d          E

76   Base     DCH      Delete Character      76   b    f    0x151         E
76   Shift    DCH      Delete Character      76   s    f    0x151         E
76   Ctrl     PFK      142                   76   c    f    0x8e          E
76   Alt      DL       Delete Line           76   a    f    0x153         E

77   Not available on keyboard

78   Not available on keyboard

79   Base     CUB      Cursor Back           79   b    f    0x104         E
79   Shift    PFK      158                   79   s    f    0x9e          E
79   Ctrl     PFK      159                   79   c    f    0x9f          E
79   Alt      PFK      160                   79   a    f    0xa0          E

80   Base     HOME                           80   b    f    0x108         E
80   Shift    PFK      143                   80   s    f    0x8f          E
80   Ctrl     PFK      144                   80   c    f    0x90          E
80   Alt      PFK      145                   80   a    f    0x91          E

81   Base     PFK      146                   81   b    f    0x92          E
81   Shift    PFK      147                   81   s    f    0x93          E
81   Ctrl     PFK      148                   81   c    f    0x94          E
81   Alt      PFK      149                   81   a    f    0x95          E

82   Not available on keyboard

83   Base     CUU      Cursor Up             83   b    f    0x101         E
83   Shift    PFK      161                   83   s    f    0xa1          E
83   Ctrl     PFK      162                   83   c    f    0xa2          E
```

83	Alt	PFK	163	83	a	f	0xa3	E
84	Base	CUD	Cursor Down	84	b	f	0x102	E
84	Shift	PFK	164	84	s	f	0xa4	E
84	Ctrl	PFK	165	84	c	f	0xa5	E
84	Alt	PFK	166	84	a	f	0xa6	E
85	Base	PFK	150	85	b	f	0x96	E
85	Shift	PFK	151	85	s	f	0x97	E
85	Ctrl	PFK	152	85	c	f	0x98	E
85	Alt	PFK	153	85	a	f	0x99	E
86	Base	PFK	154	86	b	f	0x9a	E
86	Shift	PFK	155	86	s	f	0x9b	E
86	Ctrl	PFK	156	86	c	f	0x9c	E
86	Alt	PFK	157	86	a	f	0x9d	E
87	Not available on keyboard							
88	Not available on keyboard							
89	Base	CUF	Cursor Forward	89	b	f	0x103	E
89	Shift	PFK	167	89	s	f	0xa7	E
89	Ctrl	PFK	168	89	c	f	0xa8	E
89	Alt	PFK	169	89	a	f	0xa9	E
90	Base	NUM LOCK		None				N
90	Shift	NUM LOCK		None				N
90	Ctrl	DC3	Device Control 3	90	c	c	< 19	G
90	Alt	PFK	170	90	a	f	0xaa	E
91	Base	+	Upper Left Corner	91	b	c	< 218	G
91	Shift	7	Seven	91	s	c	< 55	G
91	Ctrl	PFK	172	91	c	f	0xac	E
91	Alt	Alt+Num Entry		None				F
92	Base	+	Left Edge Int.	92	b	c	< 195	G
92	Shift	4	Four	92	s	c	< 52	G
92	Ctrl	PFK	174	92	c	f	0xae	E
92	Alt	Alt+Num Entry		None				F
93	Base	+	Lower Left Corner	93	b	c	< 192	G
93	Shift	1	One	93	s	c	< 49	G
93	Ctrl	PFK	176	93	c	f	0xb0	E
93	Alt	Alt+Num Entry		None				F
94	Not available on keyboard							
95	Base	/	Slash	95	b	c	< 47	G
95	Shift	/	Slash	95	s	c	< 47	G
95	Ctrl	PFK	179	95	c	f	0xb3	E
95	Alt	PFK	180	95	a	f	0xb4	E
96	Base	-	Top Intersection	96	b	c	< 194	G
96	Shift	8	Eight	96	s	c	< 56	G
96	Ctrl	PFK	182	96	c	f	0xb6	E
96	Alt	Alt+Num Entry		None				F
97	Base	+	Center Intersection	97	b	c	< 197	G
97	Shift	5	Five	97	s	c	< 53	G

97	Ctrl	PFK	184	97	c	f	0xb8	E
97	Alt	Alt+Num	Entry	None				F
98	Base	-	Bottom Junction	98	b	c	< 193	0
98	Shift	2	Two	98	s	c	< 50	0
98	Ctrl	PFK	186	98	c	f	0xba	E
98	Alt	Alt+Num	Entry	None				F
99	Base	¦	Vertical Bar	99	b	c	< 179	0
99	Shift	0	Zero	99	s	c	< 48	0
99	Ctrl	PFK	178	99	c	f	0xb2	E
99	Alt	Alt+Num	Entry	None				F
100	Base	*	Asterisk	100	b	c	< 42	0
100	Shift	*	Asterisk	100	s	c	< 42	0
100	Ctrl	PFK	187	100	c	f	0xbb	E
100	Alt	PFK	188	100	a	f	0xbc	E
101	Base	+	Upper Right Corner	101	b	c	< 191	0
101	Shift	9	Nine	101	s	c	< 57	0
101	Ctrl	PFK	190	101	c	f	0xbe	E
101	Alt	Alt+Num	Entry	None				F
102	Base	¦	Right Edge Int.	102	b	c	< 180	0
102	Shift	6	Six	102	s	c	< 54	0
102	Ctrl	PFK	192	102	c	f	0xc0	E
102	Alt	Alt+Num	Entry	None				F
103	Base	+	Lower Right Corner	103	b	c	< 217	0
103	Shift	3	Three	103	s	c	< 51	0
103	Ctrl	PFK	194	103	c	f	0xc2	E
103	Alt	Alt+Num	Entry	None				F
104	Base	-	Horizontal Line	104	b	c	< 196	0
104	Shift	.	Period	104	s	c	< 46	0
104	Ctrl	PFK	196	104	c	f	0xc4	E
104	Alt	PFK	197	104	a	f	0xc5	E
105	Base	-	Hyphen, Minus Sign	105	b	c	< 45	0
105	Shift	-	Hyphen, Minus Sign	105	s	c	< 45	0
105	Ctrl	PFK	198	105	c	f	0xc6	E
105	Alt	PFK	199	105	a	f	0xc7	E
106	Base	+	Plus Sign	106	b	c	< 43	0
106	Shift	+	Plus Sign	106	s	c	< 43	0
106	Ctrl	PFK	200	106	c	f	0xc8	E
106	Alt	PFK	201	106	a	f	0xc9	E
107	Not available on keyboard							
108	Base	CR	Carriage Return	108	b	c	< 13	0
108	Shift	CR	Carriage Return	108	s	c	< 13	0
108	Ctrl	CR	Carriage Return	108	c	c	< 13	0
108	Alt	PFK	100	108	a	f	0x64	E
109	Not available on keyboard							
110	Base	ESC	Escape	110	b	c	< 27	0
110	Shift	PFK	120	110	s	f	0x78	E
110	Ctrl	PFK	121	110	c	f	0x79	E

110	Alt	PFK	122	110	a	f	0x7a	E
111	Not available on keyboard							
112	Base	PFK	1	112	b	f	0x01	E
112	Shift	PFK	13	112	s	f	0x0d	E
112	Ctrl	PFK	25	112	c	f	0x19	E
112	Alt	PFK	37	112	a	f	0x25	E
113	Base	PFK	2	113	b	f	0x02	E
113	Shift	PFK	14	113	s	f	0x0e	E
113	Ctrl	PFK	26	113	c	f	0x1a	E
113	Alt	PFK	38	113	a	f	0x26	E
114	Base	PFK	3	114	b	f	0x03	E
114	Shift	PFK	15	114	s	f	0x0f	E
114	Ctrl	PFK	27	114	c	f	0x1b	E
114	Alt	PFK	39	114	a	f	0x27	E
115	Base	PFK	4	115	b	f	0x04	E
115	Shift	PFK	16	115	s	f	0x10	E
115	Ctrl	PFK	28	115	c	f	0x1c	E
115	Alt	PFK	40	115	a	f	0x28	E
116	Base	PFK	5	116	b	f	0x05	E
116	Shift	PFK	17	116	s	f	0x11	E
116	Ctrl	PFK	29	116	c	f	0x1d	E
116	Alt	PFK	41	116	a	f	0x29	E
117	Base	PFK	6	117	b	f	0x06	E
117	Shift	PFK	18	117	s	f	0x12	E
117	Ctrl	PFK	30	117	c	f	0x1e	E
117	Alt	PFK	42	117	a	f	0x2a	E
118	Base	PFK	7	118	b	f	0x07	E
118	Shift	PFK	19	118	s	f	0x13	E
118	Ctrl	PFK	31	118	c	f	0x1f	E
118	Alt	PFK	43	118	a	f	0x2b	E
119	Base	PFK	8	119	b	f	0x08	E
119	Shift	PFK	20	119	s	f	0x14	E
119	Ctrl	PFK	32	119	c	f	0x20	E
119	Alt	PFK	44	119	a	f	0x2c	E
120	Base	PFK	9	120	b	f	0x09	E
120	Shift	PFK	21	120	s	f	0x15	E
120	Ctrl	PFK	33	120	c	f	0x21	E
120	Alt	PFK	45	120	a	f	0x2d	E
121	Base	PFK	10	121	b	f	0x0a	E
121	Shift	PFK	22	121	s	f	0x16	E
121	Ctrl	PFK	34	121	c	f	0x22	E
121	Alt	PFK	46	121	a	f	0x2e	E
122	Base	PFK	11	122	b	f	0x0b	E
122	Shift	PFK	23	122	s	f	0x17	E
122	Ctrl	PFK	35	122	c	f	0x23	E
122	Alt	PFK	47	122	a	f	0x2f	E
123	Base	PFK	12	123	b	f	0x0c	E

```
123  Shift    PFK      24                  123  s    f    0x18            E
123  Ctrl     PFK      36                  123  c    f    0x24            E
123  Alt      PFK      48                  123  a    f    0x30            E

124  Base     PFK      209                 124  b    f    0xd1            E
124  Shift    PFK      210                 124  s    f    0xd2            E
124  Ctrl     PFK      211                 124  c    f    0xd3            E
124  Alt      PFK      212                 124  a    f    0xd4            E

125  Base     PFK      213                 125  b    f    0xd5            E
125  Shift    PFK      214                 125  s    f    0xd6            E
125  Ctrl     PFK      215                 125  c    f    0xd7            E
125  Alt      PFK      216                 125  a    f    0xd8            E

126  Base     PFK      217                 126  b    f    0xd9            E
126  Shift    PFK      218                 126  s    f    0xda            E
126  Ctrl     DEL                          126  c    c    < 127           0
126  Alt      DEL                          126  a    c    < 127           0
```

2.5.13.2 Keystroke Control Sequences for System Functions

The following keystroke combinations cause the indicated system functions to be performed. The notation **Padn**, where **n** is a digit, indicates the **n** key on the numeric keypad to the right of the main keyboard area.

Note: Unless otherwise noted, the functions initiated by a three-key **Ctrl-Alt-key** sequence require the **Alt** key on the **left** side of the standard PS/2 keyboard. Functions initiated with **Alt-key** (or **Shift-key**) can be selected with either the left or the right **Alt** key (or **Shift** key).

AIX System Functions

Alt-Pause Sends the interrupt signal, **SIGINT**, to all AIX processes associated with the terminal (or virtual terminal) from which this key sequence is entered. This causes most processes to terminate, although a process can arrange to ignore or take other action on this signal.

Ctrl-V Sends the quit signal, **SIGQUIT**, to all AIX processes associated with the terminal (or virtual terminal) from which this key sequence is entered. This causes most processes to terminate and produce a process image file in the current directory named **core**. However, a process can arrange to ignore or take other action on this signal.

See "termio" in topic 2.5.28 for additional AIX keystroke control sequences.

Virtual Terminal Functions

Alt-Action Changes the active display screen to the next virtual terminal (if any).

Shift-Action Changes the active display screen to the previous virtual terminal (if any).

Ctrl-Action Changes the active display screen to the command virtual terminal (if defined).

IPL (System Restart) Functions

Ctrl-Alt-Pause Performs a software reset and reboot of the system. This should only be done as a way to get out of hang conditions because the file system is not synced and data can be lost.

System Dump Functions

Ctrl-Alt-Pad7 Performs a dump of all real memory. The dump is placed into the user's dump minidisk which was set up by the installation procedure. The system is then restarted as if **Ctrl-Alt-Pause** had been pressed.

Diagnostic Functions

Ctrl-Alt-Pad4 Invokes the AIX PS/2 kernel debugger.

Related Information

In this book: "data stream" in topic 2.4.3, "display symbols" in topic 2.4.4, "hft" in topic 2.5.11, "Set Keyboard Map (HFSKBD)" in topic 2.5.11.8.5, "termio" in topic 2.5.28, and the kernel debugger in Appendix C, "Writing Device Drivers" in topic C.0.

AIX PS/2 Keyboard Description and Character Reference.

"Using the Dump Facilities" in *AIX Programming Tools and Interfaces*.

The **crash** command to examine system images in *AIX Operating System Commands Manual*.

2.5.14 lp

Purpose
Supports the line printer device driver.

Synopsis
#include <sys/lprio.h>

Description
The **lp** driver provides an interface to the port used by a printer. The driver is unique to AIX PS/2. If a port for a printer is not installed, an attempt to open fails. The **close** system call waits until all output completes before returning to the user. The **lp** driver allows only one process to write to a printer port at a time. If the printer port is busy, the **open** system call returns an error. However, the driver allows multiple **open** system calls to occur if they are **read-only**. Thus, the **splp** command can be run when the printer port is currently in use.

The **lp** driver interprets carriage returns, backspaces, line feeds, tabs, and form feeds depending on the modes that are set in the driver (via **splp**). The number of lines per page, columns per line, and the indent at the beginning of each line can also be selected. The defaults are set at 66 lines per page, and 80 columns per line with no indenting.

Subtopics
2.5.14.1 ioctl Operations

2.5.14.1 ioctl Operations

Syntax for the enhanced control function is:

```
#include <sys/lprio.h>
ioctl (fildes,command,arg)
  int fildes;                  /* file descriptor */
  int command;                 /* command type */
  struct LPRUDE *arg;          /* pointer to info structure */
```

The possible command types and their descriptions are:

IOCINFO Returns a structure defined in **sys/devinfo.h**, which describes the device.

IOCTYPE Returns device LPR (line printer) defined in **sys/devinfo.h**.

LPRGET Gets page length, width, and indent. This structure is defined in **sys/lprio.h**.

LPRGETA Gets the RS232 parameters. These are the values for baud rate, character size, stop bits, and parity. Refer to the **LPR232** structure and to the **termio.h** structure.

LPRGETV Gets optional line printer modes. See the following **LPRMODE** structure.

LPRGMOD Gets the AIX optional printer modes. These optional printer modes support the synchronous versus asynchronous write interface, as well as the **report all errors** versus **wait until error correction** error reporting mode. Refer to the following **OPRMODE** structure.

The **FONTINIT** flag is initially off. It is turned on by an application when a printer font has been initialized. It is turned off when an application wants fonts to be reinitialized and by the **lp** device driver when a FATAL printer error occurs.

LPRSET Sets page length, width, and indent values. This structure is defined in **sys/lprio.h**.

LPRSETA Sets the RS232 parameters. These are the values for baud rate, character size, stop bits, and parity. Refer to the LPR232 structure that follows and to the **termio.h** structure.

LPRSETV Sets optional line printer modes. See the following **LPRMODE** structure.

LPRSMOD Sets the AIX optional printer modes. These optional printer modes support the synchronous versus asynchronous write interface, as well as the **report all errors** versus **wait until error correction** reporting mode. Refer to the following **OPRMODE** structure.

The **FONTINIT** flag is initially off. It is turned on by an application when a printer font has been initialized. It is turned off when an application wants fonts to be reinitialized and by the **lp** device driver when a FATAL printer error occurs.

LPRUFLS Flushes any data currently in progress and causes the printer to

be initialized. This can be used during the course of normal print operations, or following an error indication.

LPRUGES Gets the device driver error structure (**LPRUDE**). The **arg** parameter must be specified to point to the structure.

LPRURES Resumes printing from the point of interruption following an error. If an error did not occur, then this control has no effect.

LPRGTOV Gets the current timeout value and stores it in the **lptimer** structure pointed to by the **arg** parameter. The timeout value is measured in seconds.

LPRSTOV Sets the timeout value. The **arg** parameter points to a **lptimer** structure. The timeout value must be given in seconds.

Most of these **ioctl** operations require the **arg** parameter to point to one of the following structures:

```
struct devinfo
{ char devtype; /* devtype for printer is '1' */
  char flags;
};

/* used with LPRGET, LPRSET */
struct lprio {
   int  ind;    /* indent value */
   int  col;    /* maximum character count */
   int  line;   /* maximum line count */
};

/* used with LPRGETV, LPRSETV */
struct lprmode {
   int  modes;  /* optional line printer modes */
};
/* bit definitions for the modes field in LPRMODE */
#define PLOT  01      /* if on, no interpretation of any character */
#define NOFF  0400    /* if on, simulate the form-feed function */
#define NONL  01000   /* if on, substitute carriage returns for */
                      /* any line-feeds */


#define NOTB  02000   /* if on, don't expand tabs, else simulate */
                      /* 8 position tabs with spaces */
#define NOBS  04000   /* if on, no backspaces to the printer */
#define NOCR  010000  /* if on, substitute line-feeds for any   */
                      /* carriage returns */
#define CAPS  020000  /* if on, map lower-case alphabetics */
                      /* to upper case */
#define WRAP  040000  /* if on, print characters beyond the page */
                      /* width on the next line, instead of */
                      /* truncating */

struct oprmode {
  int  flags;          /* optional line printer modes */
};
#define LPRSYNC     01 /* asynchronous is default. */
                       /* synchronous if on */
#define LPRALLERR   02 /* wait until error correction is default.*/
```

```
                        /* report all errors if on */
#define LPRFONTINIT 04 /* file initialization */
struct LPRUDE          /* device error-reporting structure */
{
  int  status;         /* error reason code */
  int  cresult;        /* current operation result  */
  int  tadapt;         /* adapter type */
  int  npio;           /* number of pending IO operations */
};

/* status values - error reason codes */

struct lpr232          /* settings for RS232 */
{
unsigned c_cflag;      /* error reason code */
};

/* used with LPRGTOV and LPRSTOV */
struct lptimer
{
   unsigned v_timeout; /* timeout value in seconds */
}
```

File
/dev/lp*

Related Information
In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "devinfo" in topic 2.3.15, and "asy" in topic 2.5.2.

The **splp** command in *AIX Operating System Commands Reference*.

2.5.15 lp

Purpose
Supports the IBM virtual line printer driver

Synopsis

```
#include<sys/devinfo.h>
#include<sys/b370/lpdev.h>
```

Description
This command is unique to AIX System/370.

The AIX/370 printer is based on the underlying VM spooled printer. Therefore, the driver is for the virtual 1403 or equivalent printer.

The special files used to write from the card punch is of the form **/dev/lp#**, where # is the device unit number.

The card printer can only be opened for writing. The record size of each card is determined by the buffer size specified to the **write** system call. When the device is closed, a corresponding close is issued to the VM punch device.

A few **IOCTL** operations are available besides the standard. **IOCTYPE** and **IOCINFO** commands. They are of the form

```
ioctl(fd, cmd, arg)
char *arg:
```

The commands are:

```
#define PRTsettag          1     /* set tag text */
#define PRTpurge           2     /* purge any previous records */
#define PRTto              3     /* set "TO userid" option */
#define PRTclass           4     /* set spool file class */
#define PRTcopy            5     /* set number of copies */
#define PRTform            6     /* set form identifier */
#define PRTdist            7     /* set distcode */
#define PRTfname           8     /* set file name text */
#define PRTxlate           9     /* load translate table */
```

Error Conditions
In addition to the errors listed in the **ioctl, open**, and **write** manual pages, system calls to this device can fail in the following circumstances:

ENXIO Attempt to read from the printer.

EIO The I/O to the printer failed during the operation.

Related Information
In this book, "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "open, openx, creat" in topic 1.2.199, and "write, writex" in topic 1.2.330.

2.5.16 mem, kmem

Purpose
Provides memory and kernel memory images.

Description
The **mem** and **kmem** files are pseudo-device driver files. They are found in all AIX systems. These device drivers provide access to system memory. These files can be used, for example, to examine or to patch the system.

The **mem** file is a special file that provides an image of the system's real (that is, physical) memory. Locating particular items in real memory requires locating page tables that map virtual addresses into real addresses.

The file **kmem** is similar to **mem**, except it provides an image of the kernel's and the calling program's virtual memory. The seek address is usually a kernel virtual address.

An invalid address used with **mem** or **kmem** causes an error to be returned.

Files

/dev/mem
/dev/kmem

2.5.17 mt

Purpose
Supports the IBM System/370 tape storage device driver.

```
# include <sys/devinfo.h>
# include <sys/b370/mt370.h>
# include <sys/b370/3480.h>
```

Description
Magnetic tapes are used primarily for backups, file archives, and other off-line storage. System/370 tapes are accessed though the raw interface only.

The particular magnetic tape interface used depends on the desired operating characteristics. The general form of the device name is **/dev/rmt#xy**, where:

is the device unit number

x indicates rewind (r), no-rewind (n), or rewind-unload (u) after the last close of the device. (f) indicates FORWARD-SPACE-FILE on open/rewind of the device.

y indicates the tape density, high (h) or medium (m).

Hence, **/dev/rmt0rh** and **/dev/rmt0rm** refer to the same drive, but differ in the density written. The **mt** special file is unique to AIX/370.

When opened for reading or writing, the tape is assumed to be positioned as desired. When the tape opens and writes to a tape file, a single tape mark is written if the file is no rewind on close, while a double tape mark is written if the tape is to be rewound. If the file is no rewind and opened read only, the tape is positioned after the end of file (EOF) following the data just read. Once opened, reading is restricted to between the position when opened and the next EOF. By specifically choosing **rmt** files, it is possible to read and write multiple-file tapes.

Each **read** or **write** call reads or writes the next record on the tape. The record written by **write** is the same length as the buffer given. During a **read**, record size is returned as the number of bytes read, up to the buffer size specified. Seeks are ignored. An EOF is returned as a zero-length read, with the tape positioned before the EOF.

A number of **IOCTL** operations are available. In addition to **IOCTYPE** and **IOCINFO**, the following calls are defined:

The **STIOCTOP** command issues a tape command to the appropriate device a specific number of times. The communication uses the **stop** structure:

```
struct stop {
  short st_op;        /* tape operation */
  daddr_t st_count;   /* times to perform */
};
```

The **st_op** operation is performed **st_count** times, except for commands where it is not logical to do so (rewind, for example).

The operations available are:

```
#define STWEOF  0  /* write an end-of-file record */
#define STFSF   1  /* forward space file */
#define STBSF   2  /* backward space file */
#define STFSB   3  /* Forward space block */
#define STBSB   4  /* Back space block */
#define STREW   5  /* Rewind */
#define STOFFL  6  /* Rewind and unload */
#define STNOP   7  /* NOP */
```

The status of a tape drive can be determined by issuing the MTIOCGET type **ioctl** system call.

```
/* structure for MTIOCGET - mag tape get status command */

  struct  mtget  {
    short   mt_type;   /* type of magtape device */
    short   mt_dsreg;  /* "drive status" register */
    short   mt_erreg;  /* "error" register */
    u_short mt_resid;  /* residual count */
    daddr_t mt_fileno; /* file num current position */
    daddr_t mt_blkno;  /* block number current position */
  };

/*
 * Constants for mt_type byte
 */
#define MT_ISTS           01
#define MT_ISHT           02
```

The **MTIOCLD** command issues a tape "Load Display" command (for use by IBM 3480 tape drives). The structure passed to the **MTIOCLD** command is:

```
struct ldcmd
{
       char ld_func;                /* Function code */
       char ld_msg1[LDMAXMSGLN];    /* Message 1 */
       char ld_msg2[LDMAXMSGLN];    /* Message 2 */
};
```

The message fields contain ASCII characters to be displayed as directed by the function code. Possible function codes are:

LDMOTION (Default) Maintain the message in message fields 1 and 2 until the tape drive is in motion, or the message is updated.

LDREMOVE Maintain the message in message field 1 until the tape cartridge is physically removed from the tape drive, or until the next unload/load cycle.

LDLOAD Maintain the message in message field 1 until the drive is next loaded.

LDNOOP Physically access a drive without changing the message display. This option can be used to test whether a control unit can physically communicate with a drive.

LDALL Display the message in message field 1 until a tape cartridge is physically removed from the tape drive or until the drive is next loaded. Display the message in message field 2 until the drive is next loaded.

LDSINGLE (Default) Only one of the two messages is displayed. The one which is displayed is determined by the **LDHIGH** and **LDLOW** flags.

LDDOUBLE Both messages are displayed, alternating them on the message display.

LDBLINK The single message blinks.

LDNOBLINK (Default) The single message does not blink.

LDLOW (Default) The message in only message field 1 is displayed.

LDHIGH The message in only message field 2 is displayed (8, 9, 10, and 11 are used with **LDSINGLE**).

LDAUTOLD An automatic load request is passed from the system to the automatic load controller.

LDNOAUTOLD (Default) No automatic load request is passed.

If an **STIOCLD ioctl** is to be issued and there is no tape ready in the drive, use the **O_NDELAY** flag in the **OPEN** call. This will cause the device driver to bypass checking for drive ready, and allow **only IOCTL**s through the file descriptor.

Error Conditions
In addition to the errors listed in the "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "open, openx, creat" in topic 1.2.199, "read, readv, readx" in topic 1.2.224, and "write, writex" in topic 1.2.330, system calls to this device can fail in the following circumstances:

ENXIO The tape device is not configured.

ENXIO The tape is not attached to the virtual machine.

ENXIO The tape is not loaded.

ENXIO The tape is write-protected when trying to write.

EINVAL The count is more than the architecture imposed maximum, or a read or write on a device open with **O_NDELAY** was attempted.

EIO The I/O to the tape failed during the operation.

Related Information
In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "open, openx, creat" in topic 1.2.199, "read, readv, readx" in topic 1.2.224, "write, writex" in topic 1.2.330, and "tape" in topic 2.5.27.

See "Hardware Requirements" in *AIX/370 Planning Guide*.

2.5.18 nvram

Purpose
Provides non-volatile memory image.

Description

The **nvram** file is used to access the system non-volatile memory. It is unique to AIX PS/2. The non-volatile memory consists of two sections: 64 bytes of real-time clock, configuration and system status information and a 2K non-volatile extension. Part of the extension area is used to log errors when the system cannot make a permanent copy of errors on the disk. Information written to this device is retained after power is removed from the system.

The **nvram** file is not writable by user processes. The 2K extension is accessed via a **readx** system call with a nonzero extension parameter.

An invalid virtual address used with **nvram** causes an error to be returned.

File

/dev/nvram

2.5.19 null

Purpose
Provides a null device.

Description
The **null** file is a pseudo-device driver file with no associated hardware. It is found in all AIX systems. Data written to this file is discarded. Reads from this file always return 0 bytes. Use this file to read or write null data as required.

File
/dev/null

2.5.20 osm

Purpose
Provides the interface to AIX messages.

Description
The **osm** driver collects system messages provided by the AIX kernel and application programs. It is found in all AIX systems. These system messages are available to a daemon reading this file. System messages have two sources:

The AIX kernel provides messages by calls to the kernel **printf** routine.

Application programs open and write to this file

Operating system messages are stored in a circular buffer in the system and can be read or written using the **osm*** special files. A read from **osm*** files returns some portion of the data in the circular buffer. A write to the files adds user data to the current end of the circular buffer. Any number of users may use **osm*** files in the same instance of time.

Read operations from the **osm** file start at the current end of the circular buffer and wait for new data to be added. Read operations from the file **/dev/osm.curr** start at the beginning of the circular buffer and return 0 bytes when the current end of the buffer is reached. Read operations from the **/dev/osm.all** file start at the beginning of the circular buffer, go to the current end of the circular buffer, and wait for new data to be added.

File
/dev/osm*

Related Information
In this book: "rasconf" in topic 2.3.50.

2.5.21 pty

Purpose
Implements a pseudo-terminal device.

Synopsis

```
#include <sys/devinfo.h>
#include <sys/pty.h>
#include <sys/tty.h>
```

Description
A **pty** device is a pair of bi-directional character device drivers that implement a pseudo-terminal. It is found in all AIX systems. A pseudo-terminal can act as a keyboard and a display to existing software that uses the standard terminal device interface described in "termio" in topic 2.5.28. This is useful for a variety of applications such as a remote login facility or a windowing system.

Each pseudo-terminal (or **pty**) consists of two device drivers called a controller and a server (master and slave). The *server* or *server side* of a **pty** has a standard terminal interface that can support a login shell or other software that normally communicates with terminals. The *controller* or *controller side* of a **pty** interfaces with software that generates and receives data as if it were a user at a terminal. Data written to the controller is passed directly to the server, which is then read and processed as if entered from a keyboard. Data written to the server (as if to be displayed on a terminal screen) is passed directly to the controller.

The corresponding special files are named **/dev/ptyxy** for the controller and **/dev/ttyxy** for the server, where **x** is taken from the set p-z,A-Z and **y** is taken from the set 0-9,a-f. A 1-to-1 correspondence exists between each controller-server pair with names that end in the same two characters. For example, **/dev/ptyp0** and **/dev/ttyp0** together form a **pty**.

The **ptyunits** keyword can be set in the **sysparms** stanza in **/etc/system** to change the number of **pty**s in the system from the default (defined by the **ptyunits** stanza in **/etc/master**).

Use the **devices** command to add a **pty** to the system. For applications that require a login shell on the server side of the **pty**, configure the **pty** so the **init** program creates a login shell. For IBM remote applications such as Telnet, do not configure the **pty** to start a login shell. To determine whether a particular application requires a login shell on the server side, see the appropriate application documentation.

When using a **pty** for applications other than remote login, a program must take into account the fact that a logger process may have already issued an **open** to the server side of the **pty**. When a logger opens the server side, the **open** system call suspends the process to wait for another process to open the controller side. Use the following strategy to detect this situation:

1. Open **/dev/ptyxy**.

2. Issue an **ioctl** system call to perform the **PTYSTATUS** operation.

3. If the status indicates that the server side has already been opened, then close the **pty** controller and try a **/dev/ptyxy** device with a

different value for **xy**.

4. If the status indicates that the server side has not been opened, then open the corresponding **/dev/ttyxy** device.

Note: The server side of a **pty** can be opened multiple times, but the controller can be opened only once. Attempting to open the controller side more than once causes an error.

Subtopics
2.5.21.1 select Support
2.5.21.2 ioctl Operations

2.5.21.1 select Support

The **pty** device driver supports the **select** system call in the following manner:

Read selects are satisfied when input data is available

Write selects are satisfied when data can be accepted

Exception selects are never satisfied, or hang indefinitely if n **timeout** value is specified.

See "select" in topic 1.2.242 for more information about this system call.

2.5.21.2 ioctl Operations

The interface to the server side of the **pty** device is identical to the standard interface for terminals, which is described in "termio" in topic 2.5.28.

The controller side of the **pty** device driver supports the following **ioctl** operations. (See "ioctlx, ioctl, gtty, stty" in topic 1.2.137 for detailed information about the **ioctl** system call.)

IOCTYPE Returns the device type **DD_PSEU** to indicate that this is a pseudo-terminal device. This operation ignores the **arg** parameter.

IOCINFO Copies the **devinfo** structure for the device into the buffer pointed to by the **arg** parameter passed to **ioctl**. See "devinfo" in topic 2.3.15 for details about this structure.

PTYSTATUS Returns the state of the **pty**, which is composed of two halfwords. The upper half contains the number of opens currently outstanding against the controller, and the lower half contains the number of opens currently outstanding against the server. This operation ignores the **arg** parameter.

PTYIOR Reports the number of characters available to be read. The **arg** parameter is a pointer to an integer, into which this value is stored.

PTYIOW Reports the number of eight bit bytes on the raw and canonical queues. The **arg** parameter is a pointer to an integer, into which this value is stored.

PTYGETM Gets the current mode of the **pty**. The **arg** parameter is a pointer to an integer, into which the mode is stored. See the description **PTYSETM** for an explanation of the possible mode values.

PTYSETM Sets the current mode of the **pty**. The **arg** parameter is a pointer to an integer that contains the mode to be set. The mode is zero or more of the following values logically ORed together:

- **RAWQINT** Sends the **SIGIO** signal to the process when enough buffer space is available for writing to the **pty**.
- **OUTQINT** Sends the **SIGIO** signal to the process when data is available to be read.
- **REMOTE** Controls the flow of input to the **pty**, but does not edit the input. In other words, START and STOP (**Ctrl-S** and **Ctrl-Q**) controls are processed, but no editing is done on the data stream (such as ERASE, KILL, or ICRNL).

PTYADDM Adds to the current **pty** mode by logically ORing the specified value with the existing mode. The **arg** parameter is a pointer to an integer that contains the mode bits to be set. See the description **PTYSETM** for an explanation of the possible mode values.

PTYDELM Deletes from the current **pty** mode. The **arg** parameter is a

pointer to an integer. The bits that are set in this integer specify the mode bits to be turned off. See the description **PTYSETM** for an explanation of the possible mode values.

TIOCREMOTE A mode for the controller half of a pseudo-terminal, independent of **TIOCPKT**. This mode causes input to the pseudo-terminal to be flow-controlled and not input edited (regardless of the terminal mode). Each write to the control terminal produces a record boundary for the process reading the terminal. In normal usage, a write of data is like the data typed as a line on the terminal. A write of 0 bytes is like typing an end-of-file character. **TIOCREMOTE** can be used when doing remote line editing in a window manager, or whenever flow-controlled input is required.

TIOCPKT Enables or disables packet mode. The **arg** parameter is a pointer to an integer that contains the packet mode to be enabled or disabled. The mode is zero or more of the following values logically ORed together:

TIOCPKT_STOP
: Indicates output to the terminal is stopped with STOP (**Ctrl-S**) control.

TIOCPKT_START
: Indicates output to the terminal has restarted.

TIOCPKT_DOSTOP
: Indicates packet mode is stopped when START and STOP (**Ctrl-S** and **Ctrl-Q**) controls are processed.

TIOCPKT_NOSTOP
: Indicates packet mode is stopped if the START and STOP controls are not **Ctrl-S** and **Ctrl-Q**.

TIOCUCNTL Enable/disable a mode that allows a small number of simple user **ioctl** commands to be passed through the pseudo-terminal, using a protocol similar to that of **TIOCPKT**. The **TIOCUCNTL** and **TIOCPKT** modes are mutually exclusive. This mode is enabled from the controller side of a pseudo-terminal by specifying (by reference) a nonzero parameter and disabled by specifying (by reference) a zero parameter. Each subsequent **read** from the controller side returns data written on the server part of the pseudo-terminal preceded by a zero byte, or a single byte reflecting a user control operation on the server side. A user control command consists of a special **ioctl** operation with no data. The command is given as **UIOCCMD(n**), where **n** is a number in the range of 1-255. The operation value **n** will be received as a single byte on the next **read** from the controller side. The **ioctl UIOCCMD(0)** is a no-op that may be used to probe for the existence of this facility. As with **TIOCPKT** mode, command operations may be detected with a **select** for exceptional conditions.

Error Conditions
System calls to a **pty** device fail and set **errno** to indicate the error if one or more of the following are true:

EINVAL An invalid parameter was encountered, such as a negative number of bytes to be written.

ENXIO The **pty** cannot be opened because the **pty** number is out of range.

EIO A **read**, **write**, or **ioctl** operation was attempted that requires both sides of the **pty** to be open, making a complete connection.

EACCES An attempt was made to open the controller side of a **pty** more than once.

Files

Controller Devices /dev/ptyp0, /dev/ptyp1,..., /dev/ptypf, /dev/ptyq1,..., /dev/ptyzf, /dev/ptyA0,..., /dev/ptyZf

Server Devices /dev/ttyp0, /dev/ttyp1,..., /dev/ttypf, /dev/ttyq1,..., /dev/ttyzf, /dev/ttyA0,..., /dev/ttyZf

Related Information
In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "open, openx, creat" in topic 1.2.199, "master" in topic 2.3.32, "ports" in topic 2.3.46, "system" in topic 2.3.56, and "fcntl.h" in topic 2.4.8.

The **tn** and **telnetd** commands in *AIX TCP/IP User's Guide*.

The **devices** and **init** commands in *AIX Operating System Commands Reference*.

2.5.22 punch

Purpose
Supports the IBM virtual card punch driver.

Synopsis

```
# include <sys/devinfo.h>
# include <sys/b370/crp.h>
```

Description
While punch cards are obsolete in a UNIX environment, some VM software provides extensive file transfer and communication facilities using the concept of "virtual cards". To interface to this mechanism, AIX/370 provides a driver for a virtual card punch device.

The special files used to write from the card punch are of the form **/dev/pun#**, where # is the device unit number. They are unique to AIX/370.

The card punch can only be opened for writing. The record size of each card is determined by the buffer size specified to the **write** system call. When the device is closed, a corresponding **close** is issued to the VM punch device.

A few **IOCTL** operations are available besides the standard **IOCTYPE** and **IOCINFO** commands. They are of the form:

```
ioctl(fd, cmd, arg)
char *arg;
```

The commands are:

```
#define SPsettag      1       /* set tag text */
#define SPpurge       2       /* purge any previous records */
#define SPto          3       /* set "TO userid" option */
#define SPclass       4       /* set spool file class */
#define SPcopy        5       /* set number of copies */
#define SPform        6       /* set form identifier */
#define SPdist        7       /* set distcode       */
#define SPfname       8       /* set file name text */
```

Error Conditions
In addition to the errors listed in "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "open, openx, creat" in topic 1.2.199, and "write, writex" in topic 1.2.330, system calls to this device can fail in the following circumstances:

ENXIO Attempt to read from the punch.

ENXIO The punch is not currently closed.

EIO The I/O to the punch failed during the operation.

Related Information
In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "open, openx, creat" in topic 1.2.199, and "write, writex" in topic 1.2.330.

2.5.23 reader

Purpose
Supports the IBM virtual card reader driver.

Synopsis

```
# include <sys/devinfo.h>
# include <sys/b370/crp.h>
```

Description
While punch cards are obsolete in a UNIX environment, some VM software provides extensive file transfer and communication facilities using the concept of "virtual cards". To interface to this mechanism, AIX/370 provides a driver for a virtual card reader device.

The special files used to read from the card reader are of the form **/dev/rdr#**, where # is the device unit number. They are unique to AIX/370.

The card reader can only be opened for reading. When read, the virtual deck in the reader is read, one record per read, up to the maximum size of the buffer specified. If the device is closed before all of the cards have been read, the remaining cards are flushed.

No **IOCTL** operations are available besides the standard **IOCTYPE** and **IOCINFO** commands.

Error Conditions
In addition to the errors listed in "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "open, openx, creat" in topic 1.2.199, and "read, readv, readx" in topic 1.2.224, system calls to this device can fail in the following circumstances:

ENXIO Attempt to write to the reader.

ENXIO The reader is not currently closed.

EIO The I/O from the reader failed during the operation.

Related Information
In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "open, openx, creat" in topic 1.2.199, and "read, readv, readx" in topic 1.2.224.

2.5.24 RIC

Purpose
Supports the Realtime Interface Co-processor (**RIC**) Multiport 2.

Description
The **/dev/ric** are special files that provide an image of **RIC** memory. *Ric0* refers to the first **RIC** card, *ric1* to the second, *ric2* to the third, and *ric3* to the fourth **RIC** card.

Byte address in **/dev/ric** refer to locations in memory on the appropriate **RIC** card. References to non-existent locations cause errors to be returned.

In addition to memory image, **/dev/ric** has several **ioctl** commands which are defined in **sys/1386/ric.h**.

ICACMD Issues a command to a task on the **RIC**. The argument of this **ioctl** is a pointer to a **riccmd** structure. This **ioctl** will not return until the **RIC** task has returned an interrupt. The content of the input buffer is copied into the **arg** element of the **riccmd** structure when the interrupt is received from the **RIC** task. A command 3 is sent to the **RIC** task 0 after copying the input buffer. This **ioctl** is primarily intended for sending commands to **RCM**.

ICARESET Issues a hardware reset to the **RIC** card. This command may take up to 30 seconds to complete, as the **RIC** card will run its diagnostic tests. This command also initializes the **RCM**'s maximum number of tasks, priorities, queues, and timeouts.

ICACMDNOWAIT Performs the same function as **ICACMS**, but does not wait for an interrupt from the **RIC** task, does not copy the input buffer, and does not send a command 3 to the **RIC** task upon completion.

ANOUNCETASK Identifies a task so that other device drivers that depend on the **RIC** tasks can find the appropriate task on the **RIC** card.

ICA_STAT1 Retrieves the primary status byte of a specified task on an ARTIC card. This value is returned in the user provided argument (the first byte of the *arg* field of the **riccmd** structure; see the header file **ric.h**)

ICA_STAT2 Retrieves secondary status bytes (the length of the buffer is task dependent) of a specified task on an ARTIC card. It is issued in the main module of the application process. Data is returned via the *arg* field of the **riccmd** structure.

ICA_ASYNC Allows an application process (caller) to handle asynchronous interrupts from a particular task on an ARTIC card. It is issued by the main module of the application process.

ICA_NOASYNC Undoes the effect of **ICA_ASYNC**.

ICA_CHKSIG Allows an application process to find out which task on an ARTIC card has caused an asynchronous interrupt. This

command is normally issued by the signal handler of the application process.

ICA_QURIC Reports the number of ARTIC cards installed on a machine. Returned values include slot number, card number, and card type respectively (see the header file **ric.h**), via the *arg* field of the **riccmd** structure.

Note: If a card is bad, its type will be **UNKNOWNCARD**.

Subtopics
2.5.24.1 Supporting Commands
2.5.24.2 The ARTIC Card Memory Dump
2.5.24.3 The New ioctl System Calls
2.5.24.4 Special Considerations
2.5.24.5 Dealing With Interrupts (ARTIC card --> Application Program)
2.5.24.6 Synchronous Interrupts
2.5.24.7 Asynchronous Interrupts
2.5.24.8 Processing the Interrupts
2.5.24.9 Special Considerations

2.5.24.1 Supporting Commands

There are two commands which enable users to reset an ARTIC card and then download tasks onto it.

To reset a card, enter at the command line:

```
icareset <icareset <card_number>
```

where *card_number* is some integer *n* which is 0 for the ARTIC card in the lowest slot number, 1 for the ARTIC card in the slot next to the lowest one, etc.

To load a task on an ARTIC card, enter at the command line:

```
icaload [-v] <card_number> <task_name> <task_number> [load_option]
```

where *task_name* is the file name of a task and *task_number* is an integer between 1 and **MAXTASK** inclusively (the default value for **MAXTASK** is 16; though, it can be changed in **/etc/system**). To load a task only, enter **1** for the *load_option*; by default, the loaded task is started once it is loaded.

2.5.24.2 The ARTIC Card Memory Dump

To dump the memory content of an ARTIC card, use the **dd** command, using as the input file name **dev/ric0, /dev/ric1**, etc. The recommended way of viewing the content of the output file is with the **od** command, as illustrated by the example below:

```
dd of=/dev/ric0 of=/tmp/ric0.out
od -x /tmp/ric0.out | pg
```

Alternatively, you can open one of these files with a C program and read the memory content of the card, and then use the **printf** subroutine to format the output in octal or hexadecimal.

2.5.24.3 The New ioctl System Calls

The syntax for the new **ioctl** calls is:

```
ioctl(fd,op,&info)
```

Before an **ioctl** call, the following declarations and instructions are written:

```
int fd;
struct riccmd {
       unsigned char task;
       unsigned char cmd;         /* this structure is actually
       unsigned short arglen;        defined in ric.h          */
       unsigned char arg|30|;
} ;
struct riccmd info;

fd = open("/dev/ric00,...);

    /* Here the riccmd structure is filled
       with the needed information before
       the ioctl call is issued           */
```

The **ioctl** call is issued with the correct file descriptor, the correct **ioctl** call, and all the information needed through the **riccmd** structure. This file descriptor is used by the driver to get the board number. All the following **ioctl** calls are issued to tasks running on the board corresponding to the file descriptor created with the **open** call.

The return value of the call is -1 if an error occurs, and 0 if the call is successful. If the call requests information to be returned from the driver, this information is stored in the **arg** array of the **riccmd** structure used in the call.

The new **ioctl** operations are:

ICA_STAT1 This **ioctl** operation enables a process to read the primary status byte of a task. The user passes the task number in **info.task** and the primary status of that task is returned in the first byte of **info.arg** at the end of the **ioctl** call.

ICA_STAT2 This **ioctl** operation enables a process to read the secondary status buffer of a task. The user passes the task number in **info.task** and the length of the **arg** field in **info.arglen** (the **ARTIC** driver uses this value to determine how many bytes to return without overflowing **arg** field). The secondary status bytes are returned in **info.arg** at the end of the system call. Application processes should check **info.arglen** to determine the number of bytes of data being returned.

ICA_ASYNC This **ioctl** operation is issued with the task number in **info.task**. This call informs the driver of the following:

> The application program wants to be signaled when asynchronous interrupts come from the task and card specified. The user does not need to provide the card number because the **ARTIC** driver acquires the information through the file descriptor.

The process issuing the **ioctl** call handles the signals sent by the driver when receiving asynchronous interrupt(s) from the task.

ICA_QURIC This **ioctl** operation allows callers to determine the number of ARTIC cards installed on a machine and what their types are. Data is returned in *info.arg* and *info.arglen* specifies the number of installed ARTIC cards. To determine the length in bytes of *info.arg*, multiply *info.arglen* by three.

ICA_NOASYNC This **ioctl** operation cancels the effect of the **ICA_ASYNC ioctl** operation that was previously issued to the same task on the same card. The asynchronous interrupts caused by that task are ignored until another process issues **ICA_ASYNC**.

ICA_CHKSIG This **ioctl** operation enables the user to determine the card and task numbers responsible for generating asynchronous interrupts after receiving a signal. One or more signals may need to be processed. The task and card numbers are in the first two bytes of **info.arg** respectively at the end of the call.

The program sample below shows how the **ioctl** operations **ICACMD**, **ICA_STAT1**, and **ICA_STAT2** are used:

```
  #include <fcntl.h>
  #include <sys/ioctl.h>
  #include <ric.h>
  #include <stdio.h>
  main()
  {
      int fd, i, rval, nbcopy;
        unsigned char b0, b1, b2, b3;
      struct riccmd buf;                /*defines a structure to be used in
                                            ioctls */
      fd = open ("/dev/ric00", 0_RDWR);

      /* issue a command to task 3 and check for  */
      /* error if there is any

      buf.cmd = 0xa0;    /* command to task 3 */
      buf.task=3;
      buf.arglen = 0;   /* this command has no argument   */

      if (ioctl (fd,ICACMD, &buf) == -1)
        {
          printf ("ioctl - ICACMD failed");
        goto end;
       }

      /* look at primary status to check for error */
        buf.task=3;
      rval = ioctl (fd, ICA_STAT1, &buf);
      if (rval == -1)
       {
           printf ("ioctl (ICA_STAT1) failed");
           goto end;
       }
      /* if there is an error, look at secondary status bytes for */
```

```
     /* more details                                                  */

     if (buf.arg [0] == ERROR)
     {

          /* get the secondary status bytes   */
          buf.task = 3;
          buf.arglen = 6;             /* want only 6 bytes */
          if (ioctl (fd, ICA_STAT2, &buf) == -1)
     {
               printf ("ioctl - ICA_STAT2 failed");
               goto end;
     }

     nbcopy = buf.arglen;   /* number of bytes ioctl-ICA_STAT2 returned  */
     printf ("number of bytes returned by ioctl-ICA_STAT2 = %d", nbcopy);
     printf ("secondary status byte of task is: ");
          b0 = buf.arg [0] ; b1 = buf.arg [1];
     b2 = buf.arg[2] ; b3 = buf.arg [3];
     printf ("byte" = %x byte1 = %x byte2 = %x byte3 = %x", b0, b1, b2, b3);
    }
  end: ;
  }
```

A special character file is associated with each ARTIC card. For the first ARTIC card, there is a corresponding **/dev/ric00**. For the second card, there is a corresponding **/dev/ric01**. Each file has the file protection mode of 0x666, allowing all users to open these files for reading or writing on ARTIC card tasks.

Any process can read or write to any task on any board after the correct **open** system call has been issued.

To read data from the task input buffer of a task on a particular card, the user must open the special file associated with the card and issue the **readx** command. For example, to read data from the input buffer of a task on card one, the following sequence of system calls must be issued:

```
fd = open("/dev/ric01", ...)
readx(fd, buffer, buffer_length, task_number)
```

An **ioctl** or **writex** call must be issued by **appl.level** to inform the task that the application level has read the data. The task input buffer can then be reused by the task. This information is user-defined to establish handshaking between the application and the tasks.

The **readx** system call works in the following manner:

Note: The input buffer is a buffer created by a task. The task leaves data in this buffer for a system unit process to read.

The device driver gets the input buffer page, its offset and it length from the interface block on the ARTIC card for the task. If the buffer provided by the user is shorter than the task input buffer, **readx** fills the buffer and updates the file offset pointer accordingly. The next **readx**, which is issued to read the input buffer of the same task, copies the data at the point where the last **readx** stopped, as long as the file offset pointer is smaller than the task input buffer length. When the file offset pointer is equal or greater

than the input buffer length, EOF is reached and -1 is returned. If the buffer provided by the user is longer than the task input buffer, the buffer is filled with the data from the task input buffer. File offset pointer is also updated accordingly. The **readx** call returns the number of bytes successfully read and a value of -1 in case of error.

Although this is a character special file, the **lseek** system call can be issued at the application level to change the file offset pointer. This is done when the application process has exhausted the data in the task input buffer and it is ready to read another buffer.

2.5.24.4 Special Considerations

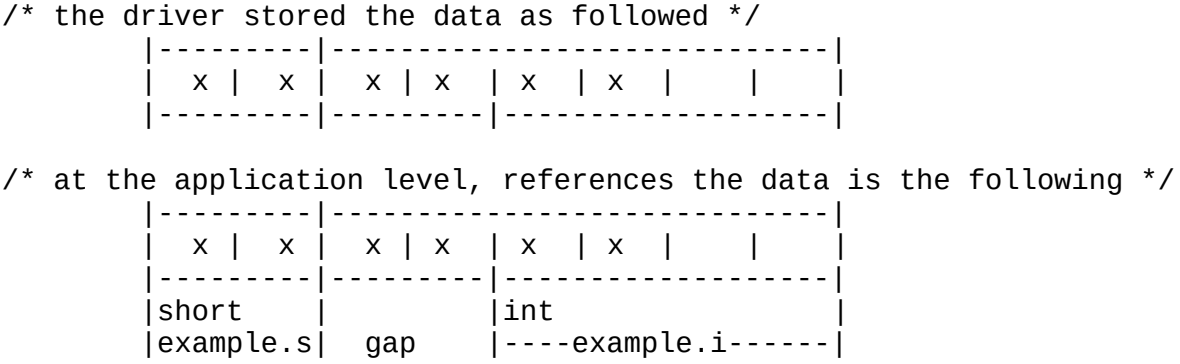
The C compiler on AIX PS/2 1.2 and the C compiler used to compile ARTIC tasks have different word alignments, as illustrated in the following example:

```
    struct example {
  short int s;
  int i;
  }
```

The size of the structure example is 6 bytes for the ARTIC compiler and 8 bytes for the AIX compiler. In AIX PS/2, the alignment for the integer is doubleword, which is 4 bytes. The variable **s** is a short **int** (2 bytes), which leaves 2 bytes of unused space between **s** and **i**.

In the case of the **readx** system call, the data is copied byte-by-byte from the input buffer of the ARTIC card task into the user buffer of type structure **struct** example. The first 6 bytes of the user buffer contain the data. At the application level, when the **i** field of structure **struct** example is referenced, only the first two bytes have valid data. The other two bytes of valid data are in the gap between the variables **s** and **i**.

The following diagram illustrates the above:



To avoid errors caused by alignment problems, all buffers used for the **readx** and **writex** system calls should only be of type **char** or unsigned **char**.

2.5.24.5 Dealing With Interrupts (ARTIC card --> Application Program)

2.5.24.6 Synchronous Interrupts

Each time a command is sent to a task with the ICACMD **ioctl** system call, the process sleeps and the **ioctl** does not return until the ARTIC task interrupts the process. This interrupt is called a ***synchronous interrupt***.

2.5.24.7 Asynchronous Interrupts

All other interrupts coming from ARTIC tasks are called ***asynchronous interrupts***. They can be interrupts coming from ARTIC tasks upon completion of an ICACMDNOWAIT **ioctl** call (from the task side, these interrupts are synchronous), or interrupts coming from the ARTIC card that request the attention of the application program.

2.5.24.8 Processing the Interrupts

The device driver processes the interrupts with its second level interrupt handler **ricintr()** in the following way:

In the case of synchronous interrupts, **ricintr()** wakes up the process sleeping in the ICACMD **ioctl** call.

In the case of asynchronous interrupts, there are two possibilities

1. Interrupts handled in the kernel by device drivers. For example, **rictty** is the device driver in the kernel that drives tty's (terminals for instance), using the ARTIC task **com232.exe** to control the transfer of data through the card.

   The kernel subroutine **icaintratch** arranges a kernel function to be called when the indicated task on the indicated on the ARTIC card (both parameters of **icaintratch** posts an interrupt to the system unit. The address of this function is put in a structure called **taskvec**. This address is checked by the ARTIC card device driver in order to know which function handles an interrupt coming from an ARTIC task.

   ```
   struct ricintrvec {
          int (*inthandler)();
          int intarg;
          int taskid;
   }

   struct ricintrvec taskvec[MAXCYCLONE][MAXTASK+1]
   /* max of cards and tasks */
   ```

   In the **rictty** thefollowing is an **icaintratch** function:

   ```
   icaintratch(0,board,ttytask[unit],rtyintr,0)

   /* ttytask[unit] is the task on the board that posts the async. int.
   /* rtyintr is the function called and it is in the taskvec structure
   ```

2. Interrupts handled by the user. If the user decides that he wants to handle asynchronous interrupts coming from certain tasks, the driver will then send a signal **sigurg** to the process that has issued an **ICA_ASYNC ioctl** call for the given tasks.

For each ARTIC card, the driver maintains a free list and a queue list. The free list is a linked list of nodes, twice as long as the number of tasks allowed to run on the card. Each time a task posts an asynchronous interrupt and there is a process which has requested to handle the interrupt coming from this task, the ARTIC card device driver, before sending the SIGURG signal to the application process, saves the card number *unit* on which the task is running, the task number, and the process id number of the process to which the driver will send the signal. The driver gets a node from the free list, stores that information in it, and attaches this node to the queue list. Each time a process issues the **ICA_CHKSG ioctl** call to find out about the nature of the interrupt, the driver returns the task number and the card number of the task that caused the interrupt to the calling process. The unnecessary node is freed from the queue list and returns to the free list. If more asynchronous interrupts occur before processes issue **ICA_CHKSIG ioctl** calls to retrieve information from the queue list, the free list may be exhausted. In this

situation, the driver reuses the node at the head of the queue. A new node is attached to the end of the queue.

The following is the structure of the linked list:

```
struct jobs_queue {
    int pid;
    unsigned char task_no ;
    unsigned char board_no;
    struct jobs_queue * next;
}
```

An opened file is either closed by the last process using it (by issuing a **close** system call) or it is closed automatically by the kernel at the end of the execution of the last process opening the file. This rule also applies to opened special files. In each case the **ricclose** function is called by the kernel. In this module, the driver reinitializes the **sig_pend_tb** table and removes the queue to return the space to the free list. These two steps are performed only for the ARTIC card being closed; other tables remain intact.

2.5.24.9 Special Considerations

In the following situation, the user decides to acknowledge asynchronous interrupts coming from certain tasks. These asynchronous interrupts are sent when this process is sleeping in a ICACMD **ioctl** call to a different task waiting for a synchronous interrupt from this task to wake it up. The situation is handled in the following manner:

1. Asynchronous interrupts that trigger signals are only processed when the process is awakened by the synchronous interrupt and returns to the user mode. Since the **tsleep** system call is issued with the PCATCH argument, if **sleep** is killed by a signal, it returns with the **TS_SIG** value. The following loop keeps the process asleep until either the synchronous interrupt wakes it, or the elapsed time exceeds 6 seconds:

   while **(tsleep(pid,...|PCATCH,6) = = TS_SIG)**

2. If more than one signal is sent to the process while it is sleeping, the device driver provides information so the user can handle all the processes. The kernel does not queue the signals when the process is sleeping but the device driver uses the two lists to keep track of signals sent during sleep.

   The information in the different buffers and primary status of a task sending multiple asynchronous interrupts during the sleep of the process is not saved by the kernel. It is up to the user to make his ARTIC tasks in a way that this transfer of information can be done in correct delays. When the process is awakened, it returns to user mode and the first signal is automatically handled in the user defined signal handler. The user then issues the **CHK_SIG ioctl** call to find out if other signals were sent to the process while sleeping. The following provides an example:

```
handler ()
{
   ...
    while ( ioctl (fd,CHECK_SIGNAL,&buf) != -1)
    {
        deal_with () /* user defined func. processes the signal */
    }
}
```

3. When the **CHK_SIG ioctl** command is issued at the application level to determine if it has any pending signals the driver searches the linked list for the first occurrence of a node with a process id that is the same as the process id of the process issuing the system call. If the search succeeds, the task number and the card number stored in this node are copied to the user buffer and this node is removed from the linked list. If the search fails, the **ioctl** call will return -1.

Related Information
The **icaload** command in the *AIX Operating System Commands Reference.*

2.5.25 st

Purpose
Supports the IBM PS/2 Internal Tape Backup Unit device driver.

Description
Magnetic tapes are used primarily for backups, file archives, and other off-line storage. The special files **rst0**, **rst4**, **rst8**, and **rst12** refer to the PS/2 Internal Tape Backup Units connected to the floppy disk controller; these files differ in the following ways:

file	**retension at open**	**rewind at close**
rst0	no	yes
rst4	no	no
rst8	yes	yes
rst12	yes	no

Only one Internal Tape Backup Unit will be recognized in a system.

The device driver maintains internal state which describes the tape position but will rewind the tape on the first open after a tape has been removed and reinserted or when a new tape is inserted. Similarly, if AIX is rebooted the tape will be rewound.

If a device is being closed and the last operation on the device was a read, the tape is not to be rewound (files **rts4** and **rst12**), and the last read did not read a filemark, then the head is effectively positioned beyond the filemark. (If the read operation was in the direction of **EOT** then the head is positioned on the **EOT** side of the filemark and vice versa.)

A number of **ioctl** operations are available, all use the following basic **ioctl** call:

```
int ioctl (fildes, command, arg)
int fildes, command;
char *arg;
```

IOCTYPE Returns **DD_TAPE** left shifted 8 bits, see **<sys/devinfo.h>**; the *arg* parameter is unused.

IOCINFO Stores device information for the file specified by *fildes* into the buffer pointed to by the *arg* parameter. See **<sys/devinfo.h>** for a description of the **devinfo** structure.

MTIOCTOP This is the **ioctl** call which is used to control tape motion and other miscellaneous tape operations. The call is slightly different:

```
#include <sys/mtio.h>
int ioctl (fildes, MTIOCTOP, arg)
int fildes, command;
struct mtop {
   short mt_op;     /* operations defined below */
```

```
    daddr_t mt_count; /* how many of them */
} *arg;
```

The following are valid operations (values of *mt_op*):

MTWEOF writes *mt_count* filemarks at the current position. The device must be opened for write or the process must have an effective user ID of that of the superuser to issue this **ioctl**.

MTFSF forward positions the tape *mt_count* filemarks. (negative numbers are allowed to backspace filemarks.)

MTBSF backward positions the tape *mt_count* filemarks.

MTFSR forward skips *mt_count* records.

MTBSR backward skips *mt_count* records.

MTREW rewinds tape (the value of *mt_count* is ignored).

MTOFFL rewinds tape and positions the head on track 0 (the value of *mt_count* is ignored).

MTNOP no operation, sets status only (the value of *mt_count* is ignored).

MTRETEN skips the tape forward to *EOT*, rewinds the tape, and positions the head on track 0

(the value of *mt_count* is ignored).

MTERASE erases the filemark map and the first block of user date on the tape; invalidates the in-memory copy of the filemark map (the value of *mt_count* is ignored).

STIOCTOP This is the command value specified when the ioctls defined in **<sys/tape.h>** are used. The call is:

```
#include <sys/tape.h>
int ioctl (fildes, STIOCTOP, arg)
int fildes, commands;
struct stop {
  short st_op;     /* operations defined below  */
  daddr_t st_count;     /* how many of them  */
} *arg;
```

The various value of *st_op* are mapped directly to **MTIOCTOP** operations as follows:

STRESET MTOFFL

STREW	**MTREW**
STERASE	**MTERASE**
STRETEN	**MTRETEN**
STWEOF	**MTWEOF**
STFSF	**MTFSF**
STFSR	**MTFSR**
STRAS1	**MTNOP**
STRAS2	**MTNOP**
STRAS3	**MTNOP**

MTIOCGET Stores device information for the file specified by *fildes* into the buffer pointed to by the *arg* parameter. See **<sys/mtio.h>** for a description of the **mtget** structure.

Note: All fields of **mtget** are set to zero on return from this **ioctl**.

STIOCGET Stores device information for the file specified by *fildes* into the buffer pointed to by the *arg* parameter. See **<sys/tape.h>** for a description of the **stget** structure.

Note: All fields of **stget** are set to zero on return from this **ioctl**.

ITGETTYPE Stores drive type / tape type for the file specified by *fildes* into the buffer pointed to by the *arg* parameter. The call is:

```
#include <sys/mtio.h>
int ioctl (fildes, ITGETTYPE, arg)
int fildes, command;
int *arg;
```

ITGETHEADER Reads the tape header for the file specified by *fildes* and stores it into the buffer pointed to by the *arg* parameter. The call is:

```
#include <sys/mtio.h>
int ioctl (fildes, ITGETHEADER, arg)
int fildes, command;
struct ctpb *arg;
```

ITPUTHEADER Stores tape header data from the buffer pointed to by the *arg* parameter into an internal buffer. This data is subsequently written to the tape for the file specified by *fildes* when the device is closed. The device must be opened for write or the process must have an effective user ID of that of the superuser to issue this **ioctl**. The call is:

```
#include <sys/mtio.h>
int ioctl (fildes, ITPUTHEADER, arg)
```

```
int fildes, command;
struct ctpb *arg;
```

ITGETFMK Reads the tape filemark map for the file specified by *fildes* and stores it into the buffer pointed to by the *arg* parameter. The call is:

```
#include <sys/mtio.h>
int ioctl (fildes, ITGETFMK, arg)
int fildes, command;
struct fmmap *arg;
```

ITPUTHEADER Stores tape filemark map from the buffer pointed to by the *arg* parameter into an internal buffer. This data is subsequently written to the tape for the file specified by *fildes* when the device is closed. The device must be opened for write or the process must have an effective user ID of that of the superuser to issue this **ioctl**. The call is:

```
#include <sys/mtio.h>
int ioctl (fildes, ITPUTFMK, arg)
int fildes, command;
struct fmmap *arg;
```

Each read or write call reads or writes an integral number of 512 bytes on the tape. A zero byte count is returned when a filemark is read, but another read will fetch the first record of the next tape file (if it exists).

Filemarks have a **BOT** (beginning-of-tape) and an **EOT** (end-of-tape) side. Forward positioning to a filemark will leave the head on the **EOT** side of the filemark, while reverse positioning will leave the head on the **BOT** side. Reading one record, skipping forward one filemark (or block) from the **BOT** side of a filemark will leave the head on the **EOT** side of the same filemark (and skipping back one filemark, or block, from the **EOT** side of a filemark will leave the head on the **BOT** side of the same filemark).

The files **st0**, **st4**, **st8**, and **st12** are block devices that behave similarly to the raw devices but use kernel buffering. They are not normally used for backup or archival purposes.

Subtopics
2.5.25.1 Using BACKUP, CPIO, TAR and TCTL
2.5.25.2 Internals
2.5.25.3 Error Conditions

2.5.25.1 Using BACKUP, CPIO, TAR and TCTL

When using the backup command the *-C* option, *-s* option and the *-1* option are not supported.

When using the tar command the *-u* and *-r* options are not supported.

The **tctl** command may be used with the Internal Tape Backup Unit (ITBU). New tapes or tapes that have not been used recently should be retensioned before using the tctl command.

The *ras1* and *ras2* options are not supported. The *erase* option erases only the first block of user data and the filemark map.

2.5.25.2 Internals

Tapes must both be servo written and formatted prior to use. Preformatted tapes are available. There is no facility for formatting tapes using the AIX PS/2 Internal Tape Backup Unit Device Driver.

The PS/2 Internal Tape Backup Unit is unique in that it has a fixed number of 32K blocks. The first block on the tape holds data about the tape format and the bad block map. A filemark map is held on the second block of the tape. Issuing an **STERASE ioctl** or issuing a **tctl** *erase* command will erase the filemark map. If an attempt is made to write too many filemarks to the tape, the driver will return **ENOSPC**. Data starts on fourth block of the tape. The third block is presently not used. It is not possible to directly read or write the first three information blocks on the tape. All reads and writes start on the fourth block.

The device driver generates ECC (Error Correction Code) which is used to regenerate lost data due to undetected errors in writing or damage to the tape. The drive does not do read after write for performance reasons and because, with the ECC, good error rates can be achieved without it.

Because of the need to generate ECC data, user data must be buffered within the driver. This is also done for performance reasons and makes it possible to keep the tape streaming.

2.5.25.3 Error Conditions

Error messages are sent on almost all failed commands to **/dev/osm**. All error messages from the AIX PS/2 Internal Tape Backup Unit Device Driver are prefixed by "Internal Tape:". Many non-fatal error messages are logged, including all blocks on which ECC decoding was necessary because of a CRC error in the data. Fatal errors cause system calls to return -1 and the type of error is available to the application program in errno. Less serious errors, such as no tape in the drive, are written only to the user's terminal. They are not logged to **/dev/osm**.

In addition to those errors listed in **ioctl**, **open**, **read**, and **write**, system calls against this device fail in the following circumstances:

ENXIO Invalid minor device number, or Internal Tape Backup Unit not present.

EBUSY Internal Tape Backup Unit is currently in use.

EIO No tape is installed in Internal Tape Backup Unit, or the Internal Tape Backup Unit is not responding to commands, or an unformatted tape is in the drive, or the tape has too many data errors.

EWRPROTECT A write protected cartridge is installed, or a tape with a format which cannot be written is installed.

EFAULT An invalid address was passed as an argument to a system call, or an invalid read or write length (a non-multiple of 512 bytes) was given.

EINVAL An invalid **ioctl** request was given.

ENOMEM Not enough memory could be allocated to perform the requested action.

ENOSPC A read or write write attempted to read/write beyond the end of tape, or an attempt to write a filemark failed because the filemark map was full.

EACCES An attempt was made to issue an **ioctl** to erase, write the header, or write the filemark map when the device was not opened for write or the process did not have an effective user ID of that of the superuser.

Files

/dev/st*
/dev/rst*

Related Information
In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "open, openx, creat" in topic 1.2.199, "close, closex" in topic 1.2.48, "read, readv, readx" in topic 1.2.224, and : hdref refid=write..

The **backup**, **cpio**, **devices**, **restore**, **tar**, and **tctl** commands in the *AIX Operating System Commands Reference*.

2.5.26 swap

Purpose
Provides interface to kernel swap space.

Description
The **swap** driver provides access to all configured swap partitions.

The **swap** driver uses the extended argument of the **readx** system call to select a partition, since more than one may be configured.

Files
/dev/swap

Related Information
In this book: "read, readv, readx" in topic 1.2.224 and "swapctl" in topic 1.2.293.

2.5.27 tape

Purpose
Supports the AIX PS/2 sequential access bulk storage medium device driver.

Description
Magnetic tapes are used primarily for backups, file archives, and other off-line storage. Tapes are accessed through the special files **mt0** and **rmt0**, which are unique to AIX PS/2. The **r** indicates "raw" which means access through the character special interface. Although streaming tape does not lend itself well to the category of a block device, a block device interface is provided to buffer I/O through the system buffer cache. This provides improved throughput when doing only sequential writes or only sequential reads, as in backup/restore. The number following the **mt** or **rmt** is the minor device number. The two low-order bits of the minor device number select the transport. If the third bit (04 octal or 0x04) is set, the driver does not rewind the tape after it is closed. If the fourth bit (010 octal or 0x08) is set, the tape is retensioned (wound completely forward and then rewound) after it is opened and before any other operations.

On a system with a single tape drive, **/dev/rmt0** does not retension the tape, but does rewind it on close. **/dev/rmt4** (bits = 0100) does not perform any special actions on open or close. **/dev/rmt8** (bits = 1000) retensions the tape and rewinds it on close; and **/dev/rmt12** (bits = 1100) retensions the tape on open, but does not rewind.

When opened for reading or writing, the tape is assumed to be positioned as desired. When the tape opens and writes to a file, a single tape mark is written if the file is no rewind on close, while a double tape mark is written if the tape is to be rewound. If the file is no rewind and opened read only, the tape is positioned after the end of file (EOF) following the data just read. Once opened, reading is restricted to between the position when opened and the next EOF. By specifically choosing **rmt** files, it is possible to read and write multiple-file tapes.

Each **read** or **write** call reads or writes the next record on the tape. The record written by **write** is the same length as the buffer given. During a **read**, the record size is returned as the number of bytes read, up to the buffer size specified. Seeks are ignored. An EOF is returned as a zero-length read, with the tape positioned before the EOF.

A number of **ioctl** operations are available. In addition to **IOCTYPE** and **IOCINFO** types, the following **ioctl** calls are defined.

The parameter to the **ioctl** system call using the **STIOCTOP** command is the address of a **stop** structure, which contains the following members:

```
short   st_op;      /* Streaming tape operation */
daddr_t st_count;  /* Number of times to perform */
```

The **st_op** operation is performed **st_count** times, except where it is not logical to do so, rewind, as an example. The operations available are:

```
#define STRESET 5   /* reset device */
#define STREW 6     /* rewind */
#define STERASE 7   /* erase tape, retension, leave at load point */
#define STRETEN 8   /* erase tape, retension, leave at load point */
#define STWEOF 10   /* write an end-of-file record */
```

```
#define STFSF  11   /* forward space file */
#define STFSR  13   /* forward space record */
#define STRAS1 15   /* drive self test 1 */
#define STRAS2 16   /* drive self test 2 */
#define STRAS3 17   /* drive self test 3 */
                    /* this test needs an */
                    /* erased write-protected tape */
```

The status of a tape drive can be determined by issuing the following STIOCGET type **ioctl** system call:

```
/* structure for STIOCGET - streaming tape get status command */
struct  stget  {
   short  st_type;      /* type of device */
  struct dsreg {
       unsigned short ds_dstat:  /* drive status */
       unsigned short ds_soft;   /* soft error count */
       unsigned short ds_under;  /* underrun count */
       unsigned char  ds_rcom;   /* command received by adapter */
       unsigned char  ds_blk;    /* adapter block count */
       unsigned char  ds_rstat;  /* status register */
       unsigned char  ds_code;   /* adapter completion code */
       unsigned char  ds_lcom;   /* last command given to adapter */
       unsigned char  ds_lstcom; /* last streaming tape device */
                                  /* drive command */
       unsigned char  ds_res[4]   /* reserved */
  } st_dsreg;
};

/*
* Constants for st_type byte - ST_SST streaming tape
*/
```

In addition to those errors listed in **ioctl**; **open**, **read**, and **write**, system calls against this device fail in the following circumstances:

EINVAL **O_APPEND** is supplied as a mode in which to open.

EINVAL A write attempt while the tape is in read mode, or a read attempt while the tape is in write mode.

EINVAL A **count** parameter to **read** or **write** is not 0, module 512.

EIO A parameter to **ioctl** is not allowed in the current streaming mode.

ENXIO The tape is write-protected or there is no tape in the drive.

Note: The streaming tape device driver has a concept of current "streaming mode". Therefore, many operations are invalid most of the time. In particular, no reads are allowed after an initial write or writes allowed after an initial read. You must wait until the device is reset either by closing a rewind-on-close special file, or by the **tctl** command.

File
/dev/rmt*

Related Information
In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137, "open, openx,

creat" in topic 1.2.199, "read, readv, readx" in topic 1.2.224, "write, writex" in topic 1.2.330, and "mt" in topic 2.5.17.

The **tctl** command in *AIX Operating System Commands Reference*.

2.5.28 termio

Purpose
Provides the general terminal interface.

Synopsis

```
#include <sys/hft.h>
#include <sys/termio.h>
#include <sys/tty.h>
```

Description
All of the asynchronous communications ports use the same general interface, regardless of the hardware used. This section discusses the common features of this interface, which is found in all AIX systems.

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, user programs seldom open these files. They are opened by **getty** and become standard input, output, and error files for a user. The first terminal file not already associated with a process group that is opened by the process group leader becomes the **control terminal** for that process group. The control terminal plays a special role in handling quit, interrupt, and suspend signals as discussed later. During a **fork** system call, the child process inherits the control terminal. A process can break the association to the group using the **setpgid** system call. The terminal may also be switched to control a different process group by using the **TIOCSPGRP ioctl**. See "BSD Compatibility" in topic 2.5.28.4.

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters can be typed at any time, even while output is occurring. These characters can be lost, however, when the input buffers become completely full or when the user accumulates the maximum number of input characters allowed that were not read by a program. Currently, this limit is 256 characters. When the input limit is reached, all the saved characters are erased from the input buffer without notice. When using the new line discipline (see "BSD Compatibility") the driver simply refuses to accept any further input and rings the terminal bell.

Normally, terminal input is processed in units of lines. A line is delimited by a new-line (ASCII LF) character, an end-of-file (ASCII EOT) character, or an end-of-line character. This means that a program attempting to read is suspended until an entire line is typed. Also, no matter how many characters are requested in the read call, at most one line is returned. It is not, however, necessary to read a whole line at once. Any number of characters can be requested in a read without losing information.

During input, erase and kill processing is performed normally. By default, the **Ctrl-H** character erases the last character typed, but does not erase beyond the beginning of the line. By default, the **Ctrl-U** character kills (deletes) the entire input line, and optionally outputs a new-line character. Both these characters operate on a keystroke basis independently of any backspacing or tabbing that was done. Both the erase and kill characters can be entered literally by preceding them with the \ (backslash) escape character. In this case, the escape character is not read. The erase and kill characters can be changed.

Certain characters have special functions on input. These functions and their default character values are summarized as follows:

EOF **Ctrl-D** or ASCII EOT is used to generate an end-of-file from a terminal. When received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line character, and the EOF is discarded. Thus, if there are not any characters waiting (indicating the EOF occurred at the beginning of a line), zero characters are passed back, which is the standard end-of-file indication.

EOL ASCII NUL is an additional line delimiter, like NL. It is not normally used.

ERASE **Ctrl-H** erases the preceding character. It does not erase beyond the start of a line, as delimited by an NL, EOF, or EOL character.

INTR **Rubout** or ASCII DEL (**Ctrl-Backspace** on the PS/2 console keyboard) generates a **SIGINT** (interrupt) signal, which is sent to all processes with the associated control terminal. Normally, each such process is forced to terminate, but arrangements can be made either to ignore the signal or to receive a trap to an agreed-upon location. See "sigaction, sigvec, signal" in topic 1.2.263.

KILL **Ctrl-U** deletes the entire line, as delimited by an NL, EOF, or EOL character.

NL ASCII LF is the normal line delimiter. It cannot be changed or escaped.

QUIT **Ctrl-V** or ACSII SYN generates a **quit** signal. Its treatment is identical to the interrupt signal except that, unless a receiving process made other arrangements, it is not only terminated but a memory file (called **core**) is created in the current working directory.

START **Ctrl-Q** or ASCII DC1 is used to resume output that was suspended by a STOP character. While output is not suspended, START characters are ignored and not read. The START/STOP characters cannot be changed or escaped.

STOP **Ctrl-S** or ASCII DC3 is used to temporarily suspend output. It is useful with terminals that have displays to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.

SUSP **Ctrl-Z** or ASCII SUB generates the signal **SIGTSTP**. If the processes in the terminal's process group do not specifically handle this signal, they are suspended. The parent process(es) are notified with a **SIGCHLD** signal and can find out about the stopped process using **wait3**. This is used for job control in shells such as **csh**. Unlike the other special characters listed here, SUSP is set using the **TIOCSLTC ioctl**. The SUSP character can only be used with the new line discipline. See "BSD Compatibility." SUSP characters cannot be escaped with a \.

The character values for INTR, QUIT, ERASE, KILL, EOF, and EOL can be changed to suit individual preferences. The ERASE, KILL, and EOF characters can be escaped by a preceding \ (backslash) character, in which case the special function is not done.

When the carrier signal from the dataset drops, a **hangup** signal (**SIGHUP**)

is sent to all processes that have this terminal as the control terminal. Unless other arrangements were made, this signal causes the process to terminate. If the hangup signal is ignored, any subsequent read returns with an end-of-file indication. Thus, programs that read a terminal and test for end-of-file can terminate appropriately.

When one or more characters are written, they are transmitted to the terminal as soon as previously written characters finish typing. Input characters are usually echoed by putting them in the output queue as they arrive. If a process produces characters more rapidly than they can be typed, it is suspended when its output queue exceeds some limit. When the output decreases to a determined threshold, the program is resumed.

Several **ioctl** system calls apply to terminal files. The primary calls use the following structures defined in the **termio.h** header file:

```
#define NCC  8
struct termio  {
     unsigned  short  c_iflag;   /* input modes */
     unsigned  short  c_oflag;   /* output modes */
     unsigned  short  c_cflag;   /* control modes */
     unsigned  short  c_lflag;   /* local modes */
     char             c_line;    /* line discipline */
     unsigned  char   c_cc[NCC];  /* control chars */
};

struct  tty_page  {
    char  tp_flags;
    unsigned char  tp_slen;
};
```

The special control characters are defined by the **c_cc** array. The relative positions and initial values for each function are as follows:

c_cc[VIN	RINTR	**Ctrl-Backspace** (DEL)
c_cc[VQU	TQUIT	**Ctrl-V** (SYN)
c_cc[VER	SERASE	**Backspace** (BS)
c_cc[VKI	LKILL	**Ctrl-U** (NAK)
c_cc[VEO	]EOF	**Ctrl-D** (EOT)
c_cc[VEO	]EOL	**Ctrl-@** (NUL)

The **c_iflag** field describes the basic terminal input control. The initial input control value is all bits clear. The possible values are:

IGNBRK	0000001	Ignore break condition.
BRKINT	0000002	Signal interrupt on break.
IGNPAR	0000004	Ignore characters with parity errors.
PARMRK	0000010	Mark parity errors.

INPCK	0000020	Enable input parity check.
ISTRIP	0000040	Strip character.
INLCR	0000100	Map new-line character (NL) to carriage return character (CR) on input.
IGNCR	0000200	Ignore carriage return character.
ICRNL	0000400	Map carriage return character to new-line character on input.
IUCLC	0001000	Maps uppercase to lowercase on input.
IXON	0002000	Enables start/stop output control.
IXANY	0004000	Enables any character to restart output.
IXOFF	0010000	Enables start/stop input control.
ASCEDIT	0020000	Enables enhanced editing on ASCII terminals.

The values in this field are described as follows:

IGNBRK If set, the break condition (a character framing error with data all zeros) is ignored. It is not put on the input queue and therefore not read by any process. Otherwise, if BRKINT is set, the break condition generates an interrupt signal and flushes both the input and output queues. If IGNPAR is set, characters with other framing and parity errors are ignored.

PARMRK If set, a character with a framing or parity error that is not ignored is read as the 3-character sequence: 0377, 0, **x**, where **x** is the data of the character received in error. If ISTRIP is not set, then a valid character of 0377 is read as 0377, 0377 to avoid ambiguity. If PARMRK is not set, a framing or parity error that is not ignored is read as the character NULL (0).

INPCK If set, input parity checking is enabled. If not set, input parity checking is disabled. This allows output parity generation without input parity errors.

ISTRIP If set, valid input characters are first stripped to 7 bits; otherwise all 8 bits are processed.

INLCR If set, a received new-line character is translated into a carriage-return character. If IGNCR is set, a received carriage-return character is ignored (not read). If ICRNL is set, a received carriage-return character is translated into a new-line character.

IUCLC If set, a received uppercase alphabetic character is translated into the corresponding lowercase character.

IXON If set, START/STOP output control is enabled. A received STOP character suspends output and a received START character restarts output. All START/STOP characters are ignored and not read. If

IXANY is set, any input character restarts output that was suspended.

IXOFF If set, the system transmits START/STOP characters when the input queue is nearly empty or full.

ASCEDIT If set, ASCII keyboards can be used to enter enhanced edit line discipline commands.

The **c_oflag** field specifies how the system treats output. The initial output control value is all bits clear.

OPOST	0000001	Postprocess output.
OLCUC	0000002	Map lowercase to uppercase on output.
ONLCR	0000004	Map new-line character to CR-NL on output.
OCRNL	0000010	Map carriage-return to new-line on output.
ONOCR	0000020	No carriage-return character output at column 0.
ONLRET	0000040	Perform carriage return function using new-line character.
OFILL	0000100	Use fill characters for delay.
OFDEL	0000200	Fill is DEL or NUL.
NLDLY	0000400	Select new-line character delays:
NL0	0	
NL1	0000400	
CRDLY	0003000	Select carriage-return delays:
CR0	0	
CR1	0001000	
CR2	0002000	
CR3	0003000	
TABDLY	0014000	Select horizontal-tab delays:
TAB0	0	
TAB1	0004000	
TAB2	0010000	
TAB3	0014000	Expand tabs to spaces.
BSDLY	0020000	Select backspace delays:

BS0	0	
BS1	0020000	
VTDLY	0040000	Select vertical-tab delays:
VT0	0	
VT1	0040000	
FFDLY	0100000	Select form-feed delays:
FF0	0	
FF1	0100000	

OPOST If set, output characters are post-processed as indicated by the remaining flags; otherwise characters are transmitted without change.

OLCUC If set, a lowercase alphabetic character is transmitted as the corresponding uppercase character. This function is often used in conjunction with IUCLC.

ONLCR If set, the new-line character is transmitted as the carriage-return new-line character pair.

OCRNL If set, the carriage-return character is transmitted as the new-line character.

ONOCR If set, no carriage-return character is transmitted when at column 0 (first position).

ONLRET If set, the new-line character is assumed to do the carriage return function. The column pointer is set to 0 and the delay specified for carriage return is used. Otherwise the new-line character is assumed to do just the line feed function; the column pointer remains unchanged. The column pointer is also set to 0 if the carriage-return character is actually transmitted.

OFILL If set, fill characters are transmitted for delay instead of a timed delay. This is useful for high baud rate terminals that need only a minimal delay.

OFDEL If set, the fill character is DEL, otherwise NUL.

NLDLY, CRDLY, TABDLY, BSDLY, VTDLY, FFDLY
The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases, a value of 0 indicates no delay. If ONLRET is set, the carriage return delays are used instead of the new-line delays.

TAB3 If set, specifies that tabs are to be expanded into spaces.

The **c_cflag** field describes the hardware control of the terminal:

CBAUD	0000017	Baud rate
B0	0	Hang up
B50	0000001	50 baud
B75	0000002	75 baud
B110	0000003	110 baud
B134	0000004	134.5 baud
B150	0000005	150 baud
B200	0000006	200 baud
B300	0000007	300 baud
B600	0000010	600 baud
B1200	0000011	1200 baud
B1800	0000012	1800 baud
B2400	0000013	2400 baud
B4800	0000014	4800 baud
B9600	0000015	9600 baud
B19200	0000016	19200 baud
EXTA	0000016	External A
EXTB	0000017	External B
CSIZE	0000060	Character size:
CS5	0	5 bits
CS6	0000020	6 bits
CS7	0000040	7 bits
CS8	0000060	8 bits
CSTOPB	0000100	Send 2 stop bits, else one.
CREAD	0000200	Enable receiver.
PARENB	0000400	Parity enable.
PARODD	0001000	Odd parity, else even.
HUPCL	0002000	Hang up on last close.
CLOCAL	0004000	Local line, else dial-up.

CBAUD These bits specify the baud rate. The zero baud rate, B0, is used

to hang up the connection. If B0 is specified, the data-terminal-ready signal is dropped. Normally, this disconnects the line. For any particular hardware, impossible speed changes are ignored.

CSIZE These bits specify the character size in bits for both transmit and receive. This size does not include the parity bit, if any. If CSTOPB is set, 2 stop bits are used; otherwise one stop bit is used. For example, at 110 baud, 2 stop bits are required.

CREAD If set, the receiver is enabled. Otherwise characters are not received.

PARENB If set, parity generation and detection is enabled and a parity bit is added to each character. If parity is enabled, the **PARODD** flag specifies odd parity if set; otherwise even parity is used.

The initial hardware control value after open is B300, CS8, CREAD, HUPCL.

HUPCL If set, the line is disconnected when the last process that has the line open, either closes it or the process terminates. That is, the data-terminal-ready signal drops.

CLOCAL If set, the line is assumed to be local, direct connection with no modem control. Otherwise modem control is assumed.

The **c_lflag** field of the parameter structure is used by the line discipline to control terminal functions. The basic line discipline (0) provides the following:

ISIG	0000001	Enable signals.
ICANON	0000002	Canonical input (erase and kill processing).
XCASE	0000004	Canonical upper/lower presentation.
ECHO	0000010	Enable echo.
ECHOE	0000020	Echo erase character as BS-SP-BS.
ECHOK	0000040	Echo new-line character after kill character.
ECHONL	0000100	Echo new-line character.
NOFLSH	0000200	Disable flushing the queue after interrupt or quit.
XSCAN	0000400	Use Scan Code Terminal Processing.

ISIG If set, each input character is checked against the special control characters INTR and QUIT. If a character matches one of these control characters, the function associated with that character is performed. If ISIG is not set, checking is not done. Thus, these special input functions are possible only if ISIG is set. These functions may be disabled individually by changing the value of the control character to an unlikely or impossible value (for example, 0377 octal or 0xFF).

ICANON If set, canonical processing is enabled. Canonical processing enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, and EOL. If ICANON is not set, then read requests are satisfied directly from the input queue. In this case, a read request is not satisfied until either at least MIN characters have been received, or the timeout value TIME has expired since the last character was received. This allows bursts of input to be read, while still allowing single-character input. The MIN and TIME values are stored in the positions for the EOF and EOL characters, respectively. The time value represents tenths of seconds.

XCASE If set along with ICANON, an uppercase letter (or the uppercase letter translated to lowercase by IUCLC) is accepted on input by preceding it with a \ (backslash) character, and is output preceded by a \ (backslash) character. In this mode, the output generates and the input accepts the following escape sequences:

For:	Use:
`	\'
|	\!
¬	\^
{	\(
}	\)
\	\\

For example, **A** is input as **\a**, **\n** as **\\n**, and **\N** as **\\\n**.

ECHO If set, characters are echoed as received. When ICANON is set, the following echo functions are possible. If ECHO and ECHOE are set, the erase character is echoed as ASCII BS SP BS, which clears the last character from a cathode-ray-tube screen. If ECHOE is set and ECHO is not set, the erase character is echoed as ASCII SP BS. If ECHOK is set, the new-line character is echoed after the kill character to emphasize that the line is deleted. Note that an escape character preceding the erase or kill character removes any special function. If ECHONL is set, the new-line character will be echoed even if ECHO is not set. This is useful for terminals set to local echo (sometimes called half duplex). Unless escaped, the EOF character is not echoed. Because EOT is the default EOF character, this prevents terminals that respond to EOT from hanging up.

NOFLSH If set, the normal flushing of the input and output queues associated with the quit and interrupt characters is not done.

XSCAN If set, Scan Code Terminal processing is performed instead of conventional processing.

Subtopics

2.5.28.1 select Support

The asynchronous terminal device driver supports the **select** system call in the following manner:

Read selects are satisfied when input data is available

Write selects are always satisfied immediately

Exception selects are never satisfied, or hang indefinitely if n **timeout** value is specified.

See "select" in topic 1.2.242 for more information about this system call.

2.5.28.2 Getting and Setting Terminal Attributes

Programs can get and set terminal attributes using the following routines:

```
#include <termios.h>

int tegetattr(fildes, termios_p)
int fildes;
struct termios * termios_p)

int tesetattr(fildes, optional_actions, termios_p)
int fildes, optional_actions;
struct termios *termios_p;
int length;
```

These routines get and set all of the supported AIX terminal attributes. AIX also provides the **TCGETA**, **TCSETA**, **TIOCGETP** and **TIOCSETP** ioctl options (described below) as alternative ways of getting and setting terminal attributes; but no one of these ioctl options supports all of the terminal attributes, and thus **tcgetattr** and **tcsetattr** are the preferred terminal interface routines.

2.5.28.3 ioctl Operations

The primary **ioctl** system calls have the format:

```
ioctl (fildes, command, arg)
int fildes;      /* file descriptor */
int command;     /* command type */
struct termio *arg;
```

The commands using this format are:

TCGETA Gets the parameters associated with the terminal and stores them in the **termio** structure referenced by **arg**.

TCSETA Sets the parameters associated with the terminal from the structure referenced by **arg**. The change is immediate.

Note: **TCGETA** and **TCSETA** do **not** get and set a complete record of the state of an HFT device. See "hft" in topic 2.5.11 for information about high-function terminal devices.

TCSETAF Waits for the output to empty, then flushes the input queue and sets the new parameters.

TCSETAW Waits for the output to empty before setting the new parameters. This form should be used when changing parameters that affect output.

The terminal paging **ioctl** calls have the format:

```
ioctl (fildes, command, arg)
int fildes;     /* file descriptor */
int command;    /* command type */
struct tty_page *arg;
```

The commands using this format are:

TCGLEN Gets the current status of the **tty_page** structure for the terminal specified as **fildes**. If paging is enabled, a value 0x1 is set in **tp_flags**. The **tp_slen** value indicates the screen length in lines.

TCSLEN Sets the status of the **tty_page** structure for this terminal. **tp_slen** means the same here as it does in **TCGLEN**. The **tp_flags** are:

PAGE_SETL	0x4	Set page length using the value in **tp_slen**.
PAGE_MSK	0x3	Command mask.
PAGE_ON	0x1	Enable paging.
PAGE_OFF	0x2	Disable paging.

Note that the **PAGE_MSK** field is interpreted as an encoding, not as separate flags.

One terminal logging **ioctl** system call has the following format:

```
ioctl (fildes, command, (char *)arg)
struct tlog *arg;
```

The **tlog** structure is defined in the **sys/termio.h** header file and contains the following members:

```
int   tl_flags
int   tl_msgqid
```

The command using this format is:

TCLOG Requests the terminal logging control functions to execute as indicated by the **tlog** structure. The **tl_flags** are:

TCLOG_ON Determines whether terminal logging is turned on or off.
TCLOG_QID Establishes message queue ID.

If **TCLOG_QID** is set, **tl_msgqid** contains the message queue ID to be used for logging from the terminal. The **tl_msgqid** value must be a message queue identifier returned from a **msgget** call.

Additional **ioctl** system calls formats are:

```
ioctl (fildes, command, arg)
int fildes;    /* file descriptor */
int command;   /* command type */
int arg;
```

The commands using this format are:

TCFLSH If **arg** is 0, flush the input queue. A value of 1 indicates flush the output queue. A value of 2 indicates flush both the input and output queues.

TCSBRK Waits for the output to empty. If **arg** is 0, then sends a break (zero bits for 0.25 seconds).

TCXONC Starts or stops control. Suspends output if **arg** is 0. Restarts suspended output if **arg** is a value of 1.

Two query **ioctl** system calls have the following format:

```
ioctl (fildes, command, &arg)
int arg;     /* returned value */
```

The commands using this format are:

TIOCNOTTY Causes the **tty** controlling this process to disassociate from the process and causes the group ID to clear. Used by background processes to free them from a controlling **tty** and process group. This **ioctl** should only be used with a file descriptor obtained from opening **/dev/tty**.

TIONREAD Gets the summation of the number of characters in the raw and canonical queues that are immediately available for reading.

Two **ioctl** system calls specific to the enhanced edit line discipline have the format:

```
ioctl (fildes, command, arg)
struct dostmplt *arg;
```

The **dostmplt** structure is defined in the **sys/termio.h** header file, and it contains the following members:

```
char *dt_tbuf
int   dt_tlen
```

The commands using this format are:

LDSETDT Sets the template buffer to contain the first **dt_tlen** characters of **dt_tbuf**, if the enhanced edit line discipline has been entered (if **c_line** equals 1, for example). At most, **DTBISIZE** characters are used. If **dt_tlen** is -1, the template buffer is not initialized.

LDGETDT Gets the current contents of the template buffer. The characters in the buffer are written starting at **dt_tbuf**, and **dt_tlen** is set to the number of characters written. At most, **DTBSIZE** characters will be returned. The characters will not be null-terminated.

2.5.28.4 BSD Compatibility

Several enhancements to the terminal interface described above are provided for compatibility with the BSD UNIX System. These enhancements ensure that most applications using the BSD TTY driver will run on the AIX PS/2 Operating System without any changes.

Subtopics

2.5.28.4.1 Line Disciplines

The system provides different **line disciplines** for controlling communications lines. There are two such disciplines available:

old The old (Version 7) terminal driver. This is sometimes used when using the standard shell **sh**.

new The standard Berkeley terminal driver, with features for job control; this must be used when using **csh**.

Line discipline switching is accomplished with the **TIOCSETD ioctl**:

```
int ldisc = LDISC;
ioctl(f, TIOCSETD, &ldisc);
```

where LDISC is **OTTYDISC** for the standard TTY driver and **NTTYDISC** for the "new" driver. The standard (currently old) TTY driver is discipline 0 by convention. Other disciplines may exist for special purposes, such as use of communications lines for network connections. The current line discipline can be obtained with the **TIOCGETD** ioctl. Pending input is discarded when the line discipline is changed.

All of the low-speed asynchronous communications ports can use any of the available line disciplines, no matter what hardware is involved. The remainder of this section discusses the "old" and "new" disciplines.

2.5.28.4.2 The Control Terminal

When a terminal file is opened, it causes the process to wait until a connection is established. In practice, user programs seldom open these files; they are opened by **getty** or **rlogind** and become a user's standard input and output file.

If a process which has no control terminal opens a terminal file, then that terminal file becomes the control terminal for that process. The control terminal is thereafter inherited by a child process during a **fork**, even if the control terminal is closed.

The file **/dev/tty** is, in each process, a synonym for a control terminal associated with that process. It is useful for programs that wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand a file name for output, when typed output is desired and it is tiresome to find out which terminal is currently in use.

A process can remove the association it has with its controlling terminal by opening the file **/dev/tty** and issuing an

```
ioctl(f, TIOCNOTTY, 0);
```

This is often desirable in server processes.

2.5.28.4.3 Process groups

Command processors such as **csh** can arbitrate the terminal between different jobs by placing related jobs in a single process group and associating this process group with the terminal. A terminal's associated process group may be set using the **TIOCSPGRP ioctl**:

```
ioctl(fildes, TIOCSPGRP, &pgrp);
```

or examined using **TIOCGPGRP**, which returns the current process group in **pgrp**. The new terminal driver aids in this arbitration by restricting access to the terminal by processes which are not in the current process group; see "Job Access Control" in topic 2.5.28.4.12.

2.5.28.4.4 Modes

The terminal drivers have three major modes, characterized by the amount of processing on the input and output characters:

cooked The normal mode. In this mode, lines of input are collected and input editing is done. The edited line is made available when it is completed by a new-line, or when the **t_brkc** character (normally undefined) or **t_eofc** character (normally an EOT, **Ctrl-D**) is entered. A carriage return is usually made synonymous with new-line in this mode, and replaced with a new-line whenever it is typed. All driver functions (input editing, interrupt generation, output processing such as delay generation and tab expansion, and so on) are available in this mode.

CBREAK This mode eliminates the character, word, and line editing input facilities, making the input character available to the user program as it is typed. Flow control, literal-next and interrupt processing are still done in this mode. Output processing is done.

RAW This mode eliminates all input processing and makes all input characters available as they are typed; no output processing is done either.

The style of input processing can also be very different when the terminal is put in non-blocking I/O mode; see the description of the **O_NONBLOCK** flag in "open, openx, creat" in topic 1.2.199 and "fcntl, flock, lockf" in topic 1.2.78. In this case, a **read** from the control terminal will never block, but rather return an error indication (EAGAIN) if there is no input available.

A process may also request that a **SIGIO** signal be sent to it whenever input is present and also whenever output queues fall below the low-water mark. To enable this mode, the **O_ASYNC** flag should be set using **fcntl**.

2.5.28.4.5 Input Editing

An AIX terminal ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely choked, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently, this limit is 256 characters. In RAW mode, the terminal driver throws away all input and output without notice when the limit is reached. In CBREAK or cooked mode, it refuses to accept any further input and, if in the new-line discipline, rings the terminal bell.

Input characters are normally accepted in either even or odd parity with the parity bit being stripped off before the character is given to the program. By clearing either the EVEN or ODD bit in the flags word, it is possible to have input characters with that parity discarded (see the "Summary of Modes" in topic 2.5.28.4.13).

In all of the line disciplines, it is possible to simulate terminal input using the **TIOCSTI ioctl**, which takes, as its third argument, the address of a character. The system pretends that this character was typed on the argument terminal, which must be the control terminal except for the superuser.

Input characters are normally echoed by putting them in an output queue as they arrive. This may be disabled by clearing the ECHO bit in the flags word using the **stty** call or the **TIOCSETN** or **TIOCSETP ioctl**s (see the "Summary of Modes" in topic 2.5.28.4.13).

In cooked mode, terminal input is processed in units of lines. A program attempting to read will normally be suspended until an entire line has been received (but see the description of **SIGTTIN** in "Job Access Control" in topic 2.5.28.4.12 and of **FIONREAD** in "Summary of Modes" in topic 2.5.28.4.13). No matter how many characters are requested in the **read** call, at most, one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a **read**, even one, without losing information.

During input, line editing is normally done, with the erase character **sg_erase** (by default, DELETE) logically erasing the last character typed and the **sg_kill** character (default, **Ctrl-U**) logically erasing the entire current input line. These characters never erase beyond the beginning of the current input line or an EOF. These characters may be entered literally by preceding them with '\'; the '\' will normally be erased when the character is typed.

The drivers normally treat either a carriage return or a new-line character as terminating an input line, replacing the return with a new-line and echoing a return and a line feed. If the CRMOD bit is cleared in the local mode word, then the processing for carriage return is disabled, and it is simply echoed as a return, and does not terminate cooked mode input.

In the new driver there is a literal-next character (normally undefined) which can be typed in both cooked and CBREAK mode preceding any character to prevent its special meaning to the terminal handler. This is to be preferred to the use of '\' escaping erase and kill characters, but '\' is retained with its old function in the new-line discipline.

The new terminal driver also provides two other editing characters in

normal mode. The word-erase character, normally **Ctrl-W**, erases the preceding word, but not any spaces before it. For the purposes of **Ctrl-W**, a word is defined as a sequence of non-blank characters, with tabs counted as blanks. Finally, the reprint character, normally **Ctrl-R**, retypes the pending input beginning on a new line. Retyping occurs automatically in cooked mode if characters which would normally be erased from the screen are fouled by program output.

2.5.28.4.6 Input Echoing and Redisplay

The new terminal driver has several modes for handling the echoing of terminal input, controlled by bits in a local mode word.

Hardcopy terminals: When a hardcopy terminal is in use, the LPRTERA bit is normally set in the local mode word. Characters which are logically erased are then printed out backwards preceded by '\' and followed by '/' in this mode.

Crt terminals: When a CRT terminal is in use, the LCRTBS bit is normally set in the local mode word. The terminal driver then echoes the proper number of erase characters when input is erased; in the normal case where the erase character is a **Ctrl-H**, this causes the cursor of the terminal to back up to where it was before the logically erased character was typed. If the input has become fouled due to interspersed asynchronous output, the input is automatically retyped.

Erasing characters from a crt: When a CRT terminal is in use, the LCRTERA bit may be set to cause input to be erased from the screen with a "backspace-space-backspace" sequence when character or word deleting sequences are used. A LCRTKIL bit may be set, causing the input to be erased in this manner on line kill sequences as well.

Echoing of control characters: If the LCTLECH bit is set in the local state word, then non-printing (control) characters are normally echoed as ^X (for some X) rather than being echoed unmodified; delete is echoed as ^?.

The normal modes for use on CRT terminals are speed-dependent. At speeds less than 1200 baud, the LCRTERA and LCRTKILL processing is painfully slow, so **stty** normally just sets LCRTBS and LCTLECH; at speeds of 1200 baud or greater, all of these bits are normally set. **stty** summarizes these option settings and the use of the new terminal driver as "newcrt."

2.5.28.4.7 Output Processing

When one or more characters are written, they are actually transmitted to the terminal as soon as previously-written characters have finished typing. (As noted above, input characters are normally echoed by putting them in the output queue as they arrive.) When a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold, the program is resumed. Even parity is normally generated on output. The EOT character is not transmitted in cooked mode to prevent terminals that respond to it from hanging up; programs using RAW or CBREAK mode should be careful.

The terminal drivers provide necessary processing for cooked and CBREAK mode output including delay generation for certain special characters and parity generation. Delays are available after backspaces **Ctrl-H**, form feeds **Ctrl-L**, carriage returns **Ctrl-M**, tabs **Ctrl-I**, and new-lines **Ctrl-J**. The driver will also optionally expand tabs into spaces, where the tab stops are assumed to be set every eight columns, and optionally convert new-lines to carriage returns followed by new-line. These functions are controlled by bits in the **tty** flags word; see "Summary of Modes" in topic 2.5.28.4.13.

The terminal driver provides for mapping between upper and lowercase on terminals lacking lowercase, and for other special processing on deficient terminals.

Finally, in the new terminal driver, there is an output flush character, normally **Ctrl-O**, which sets the LFLUSHO bit in the local mode word, causing subsequent output to be flushed until it is cleared by a program or more input is typed. This character has effect in both cooked and CBREAK modes and causes pending input to be retyped if there is any pending input. An **ioctl** to flush the characters in the input and output queues, **TIOCFLUSH**, is also available.

2.5.28.4.8 Uppercase Terminals and Hazeltines

If the LCASE bit is set in the **tty** flags, then all uppercase letters are mapped into the corresponding lowercase letter. The uppercase letter may be generated by preceding it by '\'. Uppercase letters are preceded by a '\' when output. In addition, the following escape sequences can be generated on output and accepted on input:

for	'	|	~	{	}
use	\'	\!	\^	\(	\)

To deal with Hazeltine terminals, which do not understand that ~ has been made into an ASCII character, the LTILDE bit may be set in the local mode word; in this case, the character ~ will be replaced with the character ` on output.

2.5.28.4.9 Flow Control

There are two characters (the stop character, normally **Ctrl-S**, and the start character, normally **Ctrl-Q**) which cause output to be suspended and resumed respectively. Extra stop characters typed when output is already stopped have no effect, unless the start and stop characters are made the same, in which case output resumes.

A bit in the flags word may be set to put the terminal into TANDEM mode. In this mode, the system produces a stop character (default **Ctrl-S**) when the input queue is in danger of overflowing, and a start character (default **Ctrl-Q**) when the input has drained sufficiently. This mode is useful when the terminal is actually another machine that obeys those conventions.

2.5.28.4.10 Line Control and Breaks

There are several **ioctl** calls available to control the state of the terminal line. The **TIOCSBRK ioctl** will set the break bit in the hardware interface causing a break condition to exist; this can be cleared (usually after a delay with **sleep**) by **TIOCCBRK**. Break conditions in the input are reflected as a null character in RAW mode or as the interrupt character in cooked or CBREAK mode. The **TIOCCDTR ioctl** will clear the data terminal ready condition; it can be set again by **TIOCSDTR**.

When the carrier signal from the dataset drops (usually because the user has hung up his terminal) a **SIGHUP** hangup signal is sent to the processes in the distinguished process group of the terminal; this usually causes them to terminate. The **SIGHUP** can be suppressed by setting the LNOHANG bit in the local state word of the driver. Access to the terminal by other processes is then normally revoked, so any further reads will fail, and programs that read a terminal and test for end-of-file on their input will terminate appropriately.

It is possible to ask that the phone line be hung up on the last close with the **TIOCHPCL ioctl**; this is normally done on the outgoing lines and dialups.

2.5.28.4.11 Interrupt Characters

There are several characters that generate interrupts in cooked and CBREAK mode; all are sent to the processes in the control group of the terminal, as if a **TIOCGPGRP ioctl** were done to get the process group and then a **killpg** system call were done, except that these characters also flush pending input and output when typed at a terminal (for example, **TIOCFLUSH**). The characters shown here are the defaults; the field names in the structures (given below) are also shown. The characters may be changed.

Ctrl-Backspace
: **t_intrc** (ASCII DEL) generates a **SIGINT** signal. This is the normal way to stop a process which is no longer interesting, or to regain control in an interactive program.

Ctrl-V
: **t_quitc** (ASCII SYN) generates a **SIGQUIT** signal. This is used to cause a program to terminate and produce a core image, if possible, in the file **core** in the current directory.

Ctrl-Z
: **t_suspc** (ASCII SUB) generates a **SIGTSTP** signal, which is used to suspend the current process group.

Ctrl-Y
: **t_dsuspc** (ASCII EM) generates a **SIGSTSTP** signal as **Ctrl-Z** does, but the signal is sent when a program attempts to read the **Ctrl-Y**, rather than when it is typed.

2.5.28.4.12 Job Access Control

When using the new terminal driver, if a process which is not in the distinguished process group of its control terminal attempts to read from that terminal, its process group is sent a **SIGTTIN** signal. This signal normally causes the members of that process group to stop. If, however, the process is ignoring **SIGTTIN** or has **SIGTTIN** blocked, the **read** will return -1 and set **errno** to EIO.

When using the new terminal driver with the LTOSTOP bit set in the local modes, a process is prohibited from writing on its control terminal if it is not in the distinguished process group for that terminal. Processes which are blocking or ignoring **SIGTTOU** signals are excepted and allowed to produce output.

Terminal/window sizes: In order to accommodate terminals and workstations with variable-sized windows, the terminal driver provides a mechanism for obtaining and setting the current terminal size. The driver does not use this information internally, but only stores it and provides a uniform access mechanism. When the size is changed, a **SIGWINCH** signal is sent to the terminal's process group so that knowledgeable programs may detect size changes. This facility was added in 4.3BSD and is not available in earlier versions of the system.

2.5.28.4.13 Summary of Modes

Unfortunately, due to the evolution of the terminal driver, there are four different structures which contain various portions of the driver data. The first of these (**sgttyb**) contains that part of the information largely common between version 6 and version 7 UNIX systems. The second contains additional control characters added in version 7. The third is a word of local state added in 4BSD, and the fourth is another structure of special characters added for the new driver. In the future, a single structure may be made available to programs which need to access all this information; most programs need not concern themselves with all this state.

Basic modes: sgtty:

The basic **ioctl**s use the structure defined in **<sgtty.h>**:

```
struct sgttyb {
      char   sg_ispeed;
      char   sg_ospeed;
      char   sg_erase;
      char   sg_kill;
      short  sg_flags;
};
```

The **sg_ispeed** and **sg_ospeed** fields describe the input and output speeds of the device according to the following table. Impossible speed changes are ignored. Symbolic values in the table are as defined in **<sgtty.h>**.

B0	0	(hang up dataphone)
B50	1	50 baud
B75	2	75 baud
B110	3	110 baud
B134	4	134.5 baud
B150	5	150 baud
B200	6	200 baud
B300	7	300 baud
B600	8	600 baud
B1200	9	1200 baud
B1800	10	1800 baud
B2400	11	2400 baud
B4800	12	4800 baud
B9600	13	9600 baud
EXTA	14	External A
EXTB	15	External B

Code conversion and line control required for IBM 2741's (134.5 baud) must be implemented by the user's program. The half-duplex line discipline required for the 202 dataset (1200 baud) is not supplied; full-duplex 212 datasets work fine.

The **sg_erase** and **sg_kill** fields of the argument structure specify the erase and kill characters respectively. (Defaults are **Ctrl-H** and **Ctrl-U**.)

The **sg_flags** field of the argument structure contains several bits that determine the system's treatment of the terminal:

ALLDELAY	0177400	Delay algorithm selection

BSDELAY	0100000	Select backspace delays (not implemented):
BS0	0	
BS1	0100000	
VTDELAY	0040000	Select form-feed and vertical-tab delays:
FF0	0	
FF1	0040000	
CRDELAY	0030000	Select carriage-return delays:
CR0	0	
CR1	0010000	
CR2	0020000	
CR3	0030000	
TBDELAY	0006000	Select tab delays:
TAB0	0	
TAB1	0002000	
TAB2	0004000	
XTABS	0006000	
NLDELAY	0001400	Select new-line delays:
NL0	0	
NL1	0000400	
NL2	0001000	
NL3	0001400	
EVENP	0000200	Even parity allowed on input
ODDP	0000100	Odd parity allowed on input
RAW	0000040	RAW mode: wake up on all character, 8-bit interface
CRMOD	0000020	Map CR into LF; output LF as CR-LF
ECHO	0000010	Echo (full duplex)
LCASE	0000004	Map uppercase to lower on input and lower to upper on output
CBREAK	0000002	Return each character as soon as typed
TANDEM	0000001	Automatic flow control

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases, a value of 0 indicates no delay.

Backspace delays are currently ignored but might be used for Terminet 300's.

If a form-feed/vertical tab delay is specified, it lasts for about two seconds.

Carriage-return delay type 1 lasts about .08 seconds and is suitable for the Terminet 300. Delay type 2 lasts about .16 seconds and is suitable for the VT05 and the TI 700. Delay type 3 is suitable for the concept-100 and pads lines to be at least nine characters at 9600 baud.

New-line delay type 1 is dependent on the current column and is tuned for Teletype model 37's. Type 2 is useful for the VT05 and is about .10 seconds. Type 3 is unimplemented and is 0.

Tab delay type 1 is dependent on the amount of movement and is tuned to the Teletype model 37. Type 3, called XTABS, is not a delay at all but causes tabs to be replaced by the appropriate number of spaces on output.

The flags for even and odd parity control parity checking on input and generation on output in cooked and CBREAK mode (unless LPASS8 is enabled, see below). Even parity is generated on output unless ODDP is set and EVENP is clear, in which case odd parity is generated. Input characters with the wrong parity, as determined by EVENP and ODDP, are ignored in

cooked and CBREAK mode.

RAW disables all processing save output flushing with LFLUSHO; full 8 bits of input are given as soon as it is available; all 8 bits are passed on output. A break condition in the input is reported as a null character. If the input queue overflows in raw mode, all data in the input and output queues are discarded; this applies to both new and old drivers.

CRMOD causes input carriage returns to be turned into new-lines, and output and echoed new-lines to be output as a carriage return followed by a line feed.

CBREAK is a sort of half-cooked (rare?) mode. Programs can read each character as soon as typed, instead of waiting for a full line; all processing is done except the input editing: character and word erase and line kill, input reprint, and the special treatment of \ and EOT are disabled.

TANDEM mode causes the system to produce a stop character (default **Ctrl-S**) whenever the input queue is in danger of overflowing, and a start character (default **Ctrl-Q**) when the input queue has drained sufficiently. It is useful for flow control when the "terminal" is really another computer which understands the conventions.

Note: The same "stop" and "start" characters are used for both directions of flow control; the **t_stopc** character is accepted on input as the character that stops output and is produced on output as the character to stop input, and the **t_startc** character is accepted on input as the character that restarts output and is produced on output as the character to restart input.

Basic ioctls:

A large number of **ioctl** calls apply to terminals. Some have the general form:

```
#include <sgtty.h>

ioctl(fildes, code, arg)
struct sgttyb *arg;
```

The applicable codes are:

TIOCGETP Fetch the basic parameters associated with the terminal, and store in the pointed-to **sgttyb** structure.

TIOCSETP Set the parameters according to the pointed-to **sgttyb** structure. The interface delays until output is quiescent, then throws away any unread characters, before changing the modes.

TIOCSETN Set the parameters like **TIOCSETP** but do not delay or flush input. Input is not preserved, however, when changing to or from RAW.

With the following codes **arg** is ignored.

TIOCEXCL Set "exclusive-use" mode: no further opens are permitted until the file has been closed.

TIOCNXCL Turn off "exclusive-use" mode.

TIOCHPCL When the file is closed for the last time, hang up the terminal. This is useful when the line is associated with an ACU used to place outgoing calls.

With the following codes, **arg** is a pointer to an **int**.

TIOCGETD **arg** is a pointer to an **int** into which is placed the current line discipline number.

TIOCSETD **arg** is a pointer to an **int** whose value becomes the current line discipline number.

TIOCFLUSH If the **int** pointed to by **arg** has a 0 value, all characters waiting in input or output queues are flushed. Otherwise, the value of the **int** is for the FREAD and FWRITE bits defined in **<sys/file.h>**; if the FREAD bit is set, all characters waiting in input queues are flushed, and if the FWRITE bit is set, all characters waiting in output queues are flushed.

The remaining calls are not available in standard version 7 UNIX. In cases where arguments are required, they are described; **arg** should otherwise be given as 0.

TIOCSTI The argument points to a character which the system pretends had been typed on the terminal.

TIOCSBRK The break bit is set in the terminal.

TIOCCBRK The break bit is cleared.

TIOCSDTR Data terminal ready is set.

TIOCCDTR Data terminal ready is cleared.

TIOCSTOP Output is stopped as if the stop character had been typed.

TIOCSTART Output is restarted as if the start character had been typed.

TIOCGPGRP **arg** is a pointer to an **int** into which is placed the process group ID of the process group for which this terminal is the control terminal.

TIOCSPGRP **arg** is a pointer to an **int** which is the value to which the process group ID for this terminal will be set.

TIOCOUTQ Returns in the **int** pointed to by **arg** the number of characters queued for output to the terminal.

FIONREAD Returns in the **int** pointed to by **arg** the number of characters immediately readable from the argument descriptor. This works for files, pipes, and terminals.

FIONBIO Sets the terminal to be in nonblocking mode. This is equivalent to setting the **O_NONBLOCK** file flag using the **fcntl** or **open** system call.

Tchars:

The second structure associated with each terminal specifies characters that are special in both the old and new terminal interfaces.

The following structure is defined in **<sys/ioctl.h>**, which is automatically included in **<sgtty.h>**:

```
struct tchars {
 char t_intrc;    /* interrupt                */
 char t_quitc;    /* quit                     */
 char t_startc;   /* start output             */
 char t_stopc;    /* stop output              */
 char t_eofc;     /* end-of-file              */
 char t_brkc;     /* input delimiter (like nl) */
};
```

The default values for these characters are **Ctrl-Backspace** (DEL), **Ctrl-V**, **Ctrl-Q**, **Ctrl-S**, **Ctrl-D**, and 0xff. A character value of 0xff eliminates the effect of that character. The **t_brkc** character, by default 0xff, acts like a new-line in that it terminates a 'line,' is echoed, and is passed to the program. The 'stop' and 'start' characters may be the same, to produce a toggle effect. It is probably counterproductive to make other special characters (including erase and kill) identical.

The applicable **ioctl** calls are:

TIOCGETC Get the special characters and put them in the specified structure.

TIOCSETC Set the special characters to those given in the structure.

Local mode:

The third structure associated with each terminal is a local mode word. Except for the LNOHANG bit, this word is interpreted only when the new driver is in use. The bits of the local mode word are:

LCRTBS	000001	Backspace on erase rather than echoing erase
LPRTERA	000002	Printing terminal erase mode
LCRTERA	000004	Erase character echoes as backspace-space-backspace
LTILDE	000010	Convert ~ to ` on output (for Hazeltine terminals)
LMDMBUF	000020	Stop/start output when carrier drops.
LLITOUT	000040	Suppress output translations
LTOSTOP	000100	Send SIGTTOU for background output
LFLUSHO	000200	Output is being flushed
LNOHANG	000400	Don't send hangup when carrier drops
LETXACK	001000	Diablo style buffer hacking (unimplemented)
LCRTKIL	002000	BS-space-BS erase entire line on line kill
LPASS8	004000	Pass all 8 bits through on input, in any mode
LCTLECH	010000	Echo input control chars as ^X, delete as ^?
LPENDIN	020000	Retype pending input at next read or input character
LDECCTQ	040000	Only **Ctrl-Q** restarts output after **Ctrl-S**
LNOFLSH	100000	Inhibit flushing of pending I/O when an interrupt character is typed.

The applicable **ioctl** functions are:

TIOCLBIS **arg** is a pointer to an **int** whose value is a mask containing the bits to be set in the local mode word.

TIOCLBIC **arg** is a pointer to an **int** whose value is a mask containing the bits to be cleared in the local mode word.

TIOCLSET **arg** is a pointer to an **int** whose value is stored in the local mode word.

TIOCLGET **arg** is a pointer to an **int** into which the current local mode word is placed.

Local special chars:

The final control structure associated with each terminal is the **ltchars** structure which defines control characters for the new terminal driver. Its structure is:

```
  struct ltchars {
      char t_suspc;     /* stop process signal         */
      char t_dsuspc;    /* delayed stop process signal */
      char t_rprntc;    /* reprint line                */
      char t_flushc;    /* flush output (toggles)      */
      char t_werasc;    /* word erase                  */
      char t_lnextc;    /* literal next character      */
  };
```

The default values for these characters are **Ctrl-Z**, **Ctrl-Y**, **Ctrl-R**, **Ctrl-O**, **Ctrl-W**, and undefined. A value of 0xff disables the character.

The applicable **ioctl** functions are:

TIOCSLTC **args** is a pointer to an **ltchars** structure which defines the new local special characters.

TIOCGLTC **args** is a pointer to an **ltchars** structure into which is placed the current set of local special characters.

Window/terminal sizes: Each terminal has provision for storage of the current terminal or window size in a **winsize** structure, with this format:

```
  struct winsize {
      unsigned short  ws_row;      /* rows, in characters      */
      unsigned short  ws_col;      /* columns, in characters   */
      unsigned short  ws_xpixel;   /* horizontal size, pixels  */
      unsigned short  ws_ypixel;   /* vertical size, pixels    */
  };
```

A value of 0 in any field is interpreted as "undefined"; the entire structure is zeroed on final close.

The applicable **ioctl** functions are:

TIOCGWINSZ
arg is a pointer to a **struct winsize** into which is placed the current terminal or window size information.

TIOCSWINSZ
arg is a pointer to a **struct winsize** which is used to set the current terminal or window size information. If the new

information is different than the old information, a **SIGWINCH** signal is sent to the terminal's process group.

2.5.28.5 Interaction of AIX and BSD Interfaces

The information which describes the state and controls the behavior of the AIX PS/2 TTY driver is based upon the standard **termio** structure used in AIX, supplemented with structures from the 4.3BSD version of UNIX. A collection of **ioctl** functions are used to fetch and store these structures. Although the internal implementation of the TTY driver uses one structure to hold the current state of control information for each TTY port, this full state cannot be captured or modified all at once.

However, the **termio** interface (**ioctl** commands **TCGETA**, **TCSETA**, and so forth) gives access to most of the TTY driver state. For most applications, this interface is more than sufficient and using it always produces the expected results. If a **TCGETA** is used to save a copy of the control state before changes are made, a subsequent **TCSETA** of that saved copy restores the interface to its original condition for all elements of the state which are accessible through the **termio** structure. By contrast, the BSD **TIOCGETP** and **TIOCSETP ioctl**s work with **sgtty** structures that are only mapped onto the **termio** structure. Since there is less information in a **sgtty** structure than in a **termio** structure, the interface cannot be completely saved and restored with these **ioctl**s. Even if you save a copy of the structure returned by **TIOCGETP** before you make changes to the TTY control state, you may not be able to restore the original state with a **TIOCSETP** using the saved information. This limitation is a consequence of the fact that the AIX **termio** interface contains more function than the BSD **sgtty** interface.

For source code compatibility the implementation of BSD **ioctl** support is usually adequate; problems due to the differences in the **termio** and **sgtty** interfaces are rare. In fact, problems mostly arise when a user has put the TTY driver in an unusual state, such as when OPOST is off but ICANON is on. This is because certain **sgtty** operations, such as setting RAW mode, affect many **termio** flags (for example, ICANON, ISIG, MIN, TIME, OPOST). Since little history is kept in the TTY driver (only enough to ensure that the serial communication parameters of asynchronous communication ports are maintained) turning on and off certain **sgtty** modes, such as RAW mode, effectively resets the affected **termio** modes to the default values. This is not a problem if the **termio** modes were set to the default values to begin with, but if they were in an unusual state unexpected results may occur.

The basic input editing control characters accessed with the BSD **TIOCGETC** and **TIOCSETC ioctl**s are mapped onto the corresponding **termio** (**c_cc[]**) characters.

In addition to the basic **ioctl**s and structures, there are **ioctl**s and structures used to control certain advanced features of the TTY driver. These include advanced input editing (from BSD), job control signal-generating characters (also from BSD), and TTY paging (from earlier versions of AIX). Even though these features cannot be controlled with the **termio** (**TCGETA** and **TCSETA**) **ioctl**s, there are still some interactions between the BSD advanced editing features and certain **termio** modes, as described below.

The BSD local special characters (**struct ltchars**) can only be accessed (and can be completely saved and restored with) the **TIOCGLTC** and **TIOCSLTC ioctl**s.

The BSD local mode word accessed with the **TIOCLGET** and **TIOCLSET ioctl**s is a collection of bits. Setting or resetting some of these bits changes related **termio** bits (for example, LCRTERA and ECHOE are related). Some of

these bits control modes which are effected by the state of certain **termio** modes (for example, the actions implied by LCRTKIL happen only when ICANON is true), but the state of these bits cannot be changed by changing **termio** modes (just as turning ICANON on and off does not effect the setting of ECHOK). If you intend to change any bits in the local mode word other than LTILDE, LTOSTOP, LCRTKIL and LCTLECH, you should use **TCGETA** and **TCSETA** to save the **termio** modes beforehand and restore them afterwards, as well as saving and restoring the original value of the local mode word, to ensure consistency. The effect of setting LPASS8 is to set (in the **termio** interface) CSIZE to CS8, PARENB to false, and ISTRIP to false. When LPASS8 is disabled, CSIZE is set to CS7 and ISTRIP is turned on, PARENB is turned on and PARODD is turned off. The High Function Terminal is normally in LPASS8 mode.

In order to ensure the consistency of the user's TTY interface, the various portions of the TTY driver state should be saved and restored in the order shown below. You are not required to save and restore higher numbered structures which you do not intend to change, but if you wish to change a higher numbered structure you must save and restore the lower numbered structure for complete integrity. Structures and **ioctl**s numbered identically can be saved and restored independently, in any order. However, if you don't change TTY paging you need not save and restore it.

1 **termio** structure, **TCGETA**
1 TTY paging, **TCGLEN**
1 local special characters, **TIOCGLTC**
2 **sgtty** structure, **TIOCGETP** (see note below)
2 BSD editing chars, **TIOCGETC**
3 local mode word, **TIOCLGET/TIOCLSET**
2 BSD editing chars, **TIOCSETC**
2 **sgtty** structure, **TIOCSETP**
1 local special characters, **TIOCSLTC**
1 TTY paging, **TCSLEN**
1 **termio** structure, **TCSETA**

Note: Changes to the local mode word may affect the **sgtty** structure, but if you do not use the **sgtty** structure, it is sufficient to save and restore the local mode word and the **termio** structure. In general, the **sgtty** structure is a subset of the **termio** structure and they are used in similar ways by programs that only use one or the other. A program should not use both. However, if a program is being ported to AIX from a system which uses **sgtty**, and the program cannot be modified to use **termio** instead of **sgtty**, a **TCGETA** to save the **termio** values before any sgtty operations are done and a **TCSETA** to restore those values before the program exits should be added to the program. If the program is intended to leave the tty in a different state when it exits then it should be recoded to use the **termio ioctl**s.

There are several redundant **ioctl**s and/or bits in various control structures. In general, these can be used interchangeably. For example, **TIOCFLUSH** is equivalent to **TCFLSH**; HUPCL and LNOHANG are inverses. But the rules for priority of state manipulation given above must be followed for information in control structures.

Files
/dev/tty*
/usr/include/sys/ttmap.h

Related Information

In this book: "ioctlx, ioctl, gtty, stty" in topic 1.2.137 and "hft" in topic 2.5.11.

The **csh**, **getty**, and **stty** commands in *AIX Operating System Commands Reference*.

2.5.29 trace

Purpose
Supports the event-tracing device driver.

Synopsis
#include <sys/trace.h>

Description
The **/dev/unixtrace** and **/dev/appltrace** files are special files that allow event records generated within the kernel or application programs to be passed to a user program so that the activity of a driver or other system routines can be monitored for debugging purposes. They are found in all AIX systems.

The **trace** driver supports **open, close, read**, and **ioctl** system calls. The **ioctl** system call is invoked as follows:

```
#include <sys/trace.h>
ioctl(fildes, cmd, &arg);
int fildes, cmd;

struct tr_struct {
   unsigned operation; /* TRCSET or TRCRSET */
   unsigned channels;  /* enabled channels  */
   ushort entsize;     /* reserved */
   struct trace_event * bufaddr;  /* reserved */
   ulong bufsize;      /* buffer size to use */
   union {             /* reserved */
       ushort vm_quit;
       struct {
           char vmid;
           unsigned intr_on_quit : 1;
           unsigned mpx_no : 7;
       } xvf;
   } vf;
} arg;
```

Valid values of the **cmd** parameter are:

TRCSETC Sets trace parameters. If **operation** is **TRCSET**, then this command instructs the driver to use the parameters provided in structure **arg** to enable tracing. **bufsize** indicates the size of the buffer to allocate (in number of 1K blocks) and cannot be changed once it is set. The **channels** field is a bit map indicating active and inactive channels. As an example, bit 0 corresponds to channel 31, bit 1 corresponds to channel 30, and bit 31 corresponds to channel 0. If **operation** is **TRCRSET**, then this command instructs the driver to disable all tracing.

TRCGETC Returns the current status of the trace in the structure indicated by **arg**.

The records returned from the trace device are structures with the following format:

```
struct trace_event {
    time_t stamp;        /* time stamp */
    short timeext;       /* time stamp extension */
```

```
        short seqno[2];      /* two 16-bit sequence number digits */
        short hookid;        /* channel no. and trace event code */
        unsigned pid;        /* process id */
        short iodn;          /* always -1 */
        short iocn;          /* always -1 */
        union {
            char    xcdata[20];   /* more data, depending on code */
            int     xidata[5];
        } xdata;
    };
```

The following subchannels are assigned:

Channel Number	Assignment
22	Process system calls (**acct, alarm, brk, exec, fork, fstat, getgid, getgroups, getpid, getuid, kill, lockf, nice, pause, pipe, plock, profil, ptrace, reboot, setgid, setgroups, setpgrp, setuid, times, ulimit, usrinfo, wait**)
23	Directory handling system calls (**chdir, chroot, link, mknod, unlink**)
24	I/O system calls (**access, chmod, chown, close, creat, dup, fclear, fcntl, fsync, ftrunc, ioctl, lseek, open, read, umask, uname, utime, write**)
25	File system system calls (**mount, stat, sync, ustat, umount**)
26	Time system calls (**stime, time**)
27	Signal system calls (**signal, sigblock, sigpause, sigsetmask, sigstack, sigvec**)
28	Semaphore system calls (**semctl, semget, semop**)
29	Message system calls (**msgctl, msgget, msgop**)
30	Shared memory system calls (**shmctl, shmget, shmop**)
31	User-defined events

Files

/dev/unixtrace
/dev/appltrace

Related Information
In this book: "trace_on" in topic 1.2.307, "trcunix" in topic 1.2.308, and "rasconf" in topic 2.3.50.

The **trace** command in *AIX Operating System Commands Reference*.

The discussion of **trace** in *AIX Programming Tools and Interfaces*.

2.5.30 tty

Purpose
Supports the controlling terminal interface.

Synopsis

```
#include<sys/hft.h>
#include<sys/termio.h>
#include<sys/tty.h>
```

Description
For each process, the **/dev/tty** special file is a synonym for the associated control terminal. It is found in all AIX systems. This file is useful to programs or shell sequences that wish to ensure writing messages on the terminal regardless of how output is redirected. It can also be used for programs that demand the name of a file for output when typed output is desired, and to find out what terminal is currently in use.

Files

/dev/tty
/dev/tty*

Related Information
In this book: "hft" in topic 2.5.11.

2.6 Chapter 6. Advanced Display Graphics Support Library

Subtopics

2.6.1 About This Chapter

This chapter describes the Advanced Display Graphics Support Library (GSL), which is unique to AIX PS/2. This application program interfaces to various output devices.

Subroutines, located in the **libgsl.a** library, are provided by the GSL. The **gslerrno.h** header file must be included with a **#include** statement to provide return values for the GSL subroutines.

Notes:

1. All GSL parameters are passed by reference, making the subroutines compatible with FORTRAN, in which parameters are always passed by reference. All parameters are therefore passed as pointers in C and are declared as **VAR** parameters in Pascal. The name of a GSL subroutine should be followed by an _ (underscore) in C and Pascal, but not in FORTRAN for compatibility with the RT. The _ (underscore) is optional on AIX PS/2 for all three languages.

2. Applications can be linked to either a shared or an unshared version of the GSL library. Before writing an application program that uses the GSL, see "Using the GSL Libraries" in topic 2.6.4.3.

2.6.2 Overview

This overview section contains information to help you become acquainted with the terms, major concepts, and functionality of the Graphics Support Library. The first section defines special GSL terms. The next section contains descriptions of the functionality provided by the GSL and an explanation of some of the important concepts needed to use the GSL. Later sections describe the attributes, cursor operations, and coordinate clipping and transformation capabilities of the GSL. Refer to the following list of contents for the first page of each of these discussions.

Subtopics

2.6.2.1 Definitions

The following terms are defined for this chapter:

Frame buffer A display adapter frame buffer is memory storage containing a representation of a display image.

Geometric text The two types of text supported by the GSL are annotated or standard text, and geometric text, which is also referred to as a programmable character set (PCS) or stroke text. For more information on annotated text, see "fonts" in topic 2.3.19.

KSR Mode KSR (keyboard send-receive) Mode causes a virtual terminal to act like a standard ASCII terminal, with some extensions, for both input and output. See "hft" in topic 2.5.11 for more information about KSR Mode.

Monitor Mode In Monitor Mode, a virtual terminal lets an application directly access the display adapter without conflict with the standard virtual terminal output mechanism. Further information about Monitor Mode is found under "hft" in topic 2.5.11.

Pixel A pixel, or picture element, is one point in the frame buffer or on the display.

Pixel map Also known as a ***pixmap***, this is an object that defines the characteristics of a rectangle. See "gsxblt" in topic 2.6.57 for a list of the elements defined by a pixel map.

Ring Buffer A virtual terminal in Monitor Mode can share a ring buffer with an application and place data from input devices in the buffer. The ring buffer mechanism dramatically shortens the input data path from the virtual terminal to the application.

2.6.2.2 Concepts

The GSL allows applications to perform graphics operations without the need to directly manipulate the underlying hardware. The GSL also supports the display of fixed-spaced characters in text.

The GSL assumes that an application using it runs in its own virtual terminal. A virtual terminal can operate in either KSR Mode (the default) or in Monitor Mode. An application may use Monitor Mode and the ring buffer to derive its own graphics interface. The GSL provides an interface that lets a user generate graphics interactively without detailed knowledge of the display adapter and input data formats. The GSL works only with the application virtual terminal in Monitor Mode. Part of the GSL initialization is to place the virtual terminal in Monitor Mode. This forces some restrictions on the use of the display adapter. The application virtual terminal can be one of several virtual terminals opened by a user, but only one virtual terminal can be active for input at any time. Several virtual terminals can be active for output at any time if multiple displays are attached, with one virtual terminal active for output on each display. All virtual terminals but one, however, are inactive for input at a given time. The active virtual terminal for input can write to the display adapter and receive input from devices.

Applications must respond to user requests to become active or to release control of the display (become inactive). The transfer of control of the display occurs with two signals (a release request, **SIGRETRACT**, and a grant notification, **SIGGRANT**) and a write to the **HFT** device driver to acknowledge the release signal. After initialization, the GSL processes these two signals and writes to the device driver so that it can determine when it can and cannot write to the adapter. Routines that an application supplies that get called by the GSL signal handlers can be identified by the application during GSL initialization. The application can therefore respond appropriately to requests to be active or inactive.

The GSL provides a set of graphics output functions. Applications can supply additional functions that access the display adapter directly. Such an application routine can function only when the virtual terminal is active, and the virtual terminal must not become inactive while the routine is operating. The GSL provides a function that indicates to the application whether its virtual terminal is active or inactive, and if active, postpones GSL processing of the **SIGRETRACT** signal until the application has finished modifying the display. Another function causes the GSL to resume processing of the signal.

One of the GSL output functions or an application-supplied output function may be operating at the time of the **SIGRETRACT** signal. If this is the case, the function only has 30 seconds (real time) to complete the adapter operation and acknowledge the **SIGGRANT** after receiving that signal. After the 30 seconds, the **HFT** device driver sends a **SIGKILL** signal that terminates the virtual terminal. The application should be designed with this consideration in mind, or the user should be made aware of the time limit for applications that involve switching virtual terminals and have lengthy drawing operations.

The virtual terminal subsystem dictates that when a Monitor Mode virtual terminal becomes inactive and then active, the application must restore the display adapter state. At initialization the application can direct the GSL to use either of two mechanisms for restoration.

GSL Control

The GSL saves the frame buffer and the adapter state, such as the color map, at the time of the **SIGRETRACT** and restores both at the time of the **SIGGRANT**. Unfortunately, saving and restoring large frame buffers can be relatively expensive in terms of time and virtual storage space. Under this mechanism, an output operation initiated while the application virtual terminal is inactive suspends the application until its virtual terminal becomes active. If the virtual terminal is inactive when the application requests postponement of **SIGRETRACT** signal handling, the GSL suspends the application until the virtual terminal becomes active.

Application Control

The GSL saves the adapter state, but not the frame buffer, at the time of the **SIGRETRACT** request and calls an application routine (if provided) at the time of the **SIGGRANT**. This routine could process the applications data structure(s) to reconstruct the frame buffer. Under this mechanism, an output operation initiated while the application virtual terminal is inactive causes the output routine to return without writing to the display adapter. The routine returns a code indicating an invalid status in this circumstance. If the virtual terminal is inactive when the application requests postponement of **SIGRETRACT** signal handling, the GSL sends a code indicating that the application cannot access the display.

Regardless of the mechanism chosen, the GSL calls an application routine (if provided) at the time of the **SIGRETRACT** request and calls an application routine (if provided) at the time of the **SIGGRANT** notification. One or both restoration routines can be chosen for an application as appropriate.

An application cannot write to standard output (using system write) on a virtual terminal that is in Monitor Mode. However, at initialization, the GSL accepts a specified file descriptor as the Monitor Mode virtual terminal from the application, and directs output to this file descriptor. An application can use more than one virtual terminal, and the virtual terminals can be mapped to different displays simultaneously. This reserves standard output for other uses such as **dbx** and **sdb**, symbolic debuggers on the PS/2 and RT, respectively.

When the ring buffer mechanism is used for processing input, the virtual terminal places input from the keyboard, locator, LPFK, valuator, or pick device in a ring buffer shared between the application and the virtual terminal. The virtual terminal causes the generation of the **SIGMSG** signal when it places the data for an input event in an empty ring buffer. At initialization, an application can select either method. However, the GSL supports only the ring buffer mechanism to optimize performance. If used, a ring buffer must be allocated by the application and made available to the GSL at initialization. The GSL sets up the virtual terminal linkage to the buffer and sets up a signal handler to catch the **SIGMSG** signal that it uses to satisfy application requests for input.

The application must then let the GSL process the ring buffer input pointer and parse the input events by invoking the appropriate input function. Whenever the application has selected the ring buffer mechanism, the application can use GSL input to enable and disable input events.

The application can provide a signal handler to catch the **SIGMSG** signal if all of the following conditions are met:

1. The signal handler is set up after the GSL is initialized.

2. The signal handler is set up using the **SIGVEC** enhanced signal function. **SIGVEC** returns the address of the GSL signal handler.

3. The signal handler must indirectly call the GSL signal handler before doing anything else. The indirect call uses the address returned by the **SIGVEC** signal.

Enhanced signals are used to block further reporting of the signal being processed until the signal handler returns. When the signal handler returns, the signal is automatically reset and unblocked.

When keyboard events are enabled, the virtual terminal puts **all** keystrokes in the ring buffer, including those that may normally have special meaning to the operating system (such as break). The application can let the system continue processing certain keystrokes by setting the virtual terminal break map.

For further information on Monitor Mode operation, see "Monitor Mode (MOM)" in topic 2.5.11.4.2.

2.6.2.3 Attributes

A set of attributes that determine how a function works, or determine appearance characteristics on a display, govern all GSL operations affecting the frame buffer. Attributes are characteristics that do not change often and therefore do not need to be parameters for the output functions. Some common attributes govern all output operations while others are unique to a particular category of output.

Subtopics
2.6.2.3.1 Common Attributes
2.6.2.3.2 Unique Attributes

2.6.2.3.1 Common Attributes

Color display adapters may be considered to have multiple storage planes or layers forming the frame buffer, with each plane acting like the single frame buffer for a bilevel monochrome display. When writing a pixel into a multiplane frame buffer, one may write to all the planes or to a subset. The GSL ***plane mask*** attribute identifies which planes of the frame buffer the GSL functions modify.

The color of a pixel on the display is ultimately determined by the color value of the pixel stored in the frame buffer. There are ***VLT-based adapters***, in which the pixel color value serves as an index into a ***video lookup table (VLT)***. The entry in the VLT for an index contains a value for each of the red, green, and blue digital-to-analog converters (DACs) on the adapter, which drive the color guns in the display tube. The actual color resulting from a particular pixel color value (VLT index) depends on the values loaded into the VLT, which may be any values. There are also ***true color adapters*** in which the pixel color value actually drives the DACs, without the level of indirection forced by the VLT.

An application can determine the mapping from the color used in operations on the frame buffer to the actual color shown on a display by using the GSL ***color map*** attribute. For VLT-based adapters, the GSL actually loads the adapter VLT, using color values provided by the application; the **color** used by the application is really an index into the VLT. For true color adapters, the color map serves strictly as an internal mapping from the color value specified to the actual color value loaded into the frame buffer, and the **color** used by the application is an index into the mapping table.

The application may set the color map by providing an array of color specifications. The maximum number of specifications is dependent on the display adapter and is determined by the number of VLT entries, or by the number of bit planes for true color adapters. The color specification for each color index comprises three intensity values, one each for the red, green, and blue DACs. Each intensity value must range from 0 - 0x3FFF. For a VLT-based display adapter, the GSL maps the color specification to the nearest available color produced by the adapter; the GSL truncates the intensity value for a color to produce a value equal in resolution to the DAC for that color.

The ***logical operation*** attribute determines how the GSL combines the pixels it generates with the current contents of the frame buffer. Sixteen Boolean combinations exist between a source (the GSL-produced pixels) and a destination frame buffer, but can only be used with certain display adapters. The GSL does, however, ensure support for the most recognizable and useful Boolean combinations (REPLACE, AND, OR, and Exclusive-or) regardless of hardware support.

The following table shows the categories of functions to which the common attributes apply.

Figure 6-1. Categories of Functions to Which Common Attributes Apply

Area	**Output**	**Pixel Block**	**Cursor**
Plane mask	yes	yes	yes

Color map	yes	yes	yes
Logical operation	yes	no	no

2.6.2.3.2 Unique Attributes

Some unique attributes, along with the plane mask, color map, and logical operation attributes, govern the GSL functions that affect the frame buffer.

Lines

The GSL draws all lines a single pixel thick. The following unique attributes that govern line drawing can be changed:

Line style Determines the pattern appearance of the line. The line style attribute provides for solid, dashed, dotted, dashed-dotted, and dashed-dotted-dotted lines, and for line patterns defined by the application.

Line color Is an index into the color map table (or VLT).

Markers

The marker attributes determine characteristics of symbols used to mark points. The GSL provides a set of predefined markers for the application to select. A marker can be custom defined by the application.

The application may change the following unique attributes that govern marker operations:

Marker color Sets the color of a marker, and is an index into the color map table (or VLT).

Marker origin Sets the point in the marker pattern that is placed at the position indicated by the application for the polymarker subroutine.

Marker style Selects predefined or custom markers.

Marker width Defines the width of the pattern for a custom marker.

Marker height Defines the height of the pattern for a custom marker.

Marker pattern Sets the form of the custom marker, and is a bit array defined by the application.

Text

The GSL places characters with a transparent background. That is, only the strokes in a character change data in the frame buffer. These unique attributes govern text operations and can be set by an application:

Text font Sets which of the available fonts is used for the characters.

Text color Sets the color of the text and is an index into the color map table or VLT.

Code page Sets the page from which graphic symbols are drawn.

Baseline direction Sets the direction in which characters are written to the baseline for the text. The baseline for the string is placed at the location given in the command to write text.

Xtext

The GSL provides a high-performance text drawing routine. This routine draws both the foreground and background for each character. In addition, clipping and logical operations are provided. Note that, unlike the standard GSL annotated text, xtext does not provide the baseline direction attribute. Xtext is always drawn from left to right.

xtext foreground color
Sets the color of the foreground of the text and is an index into the color map table or VLT.

xtext background color
Sets the color of the background of the text and is an index in the color map table or VLT.

xtext logical operation
Determines how the GSL combines the pixels of the text with the current contents of the frame buffer.

xtext clip box Specifies a rectangular area of pixels in the frame. The **gsxtxt** subroutine will not place full or partial characters outside this rectangular area.

Filled Areas

The edges of an area are treated as part of the area and only define the area to be filled. The GSL does not treat the edges of an area as lines. The application may change the following unique attributes that govern fill operations:

Fill color Provides an index into the color map table (or VLT).

Fill pattern Specifies the identifier for the pattern used to fill the area.

Cursor

The GSL provides a single cursor for the application at any given time. The application can use either a single-color cursor or a multicolor, masked cursor. The single-color cursor provides higher performance and allows a larger cursor pattern. The multicolor cursor provides foreground and background colors, clipping, logical operations, and masking.

The application may change the unique attributes that govern cursor operations. For the single-color cursor, these attributes are:

Cursor pattern Sets the cursor shape and consists of a bit array defined by the application. The minimum and maximum sizes for the cursor pattern are device-dependent and available to the application.

Cursor color Sets the cursor color and provides an index into the color map table or VLT.

Cursor origin Sets the point in the cursor pattern that is placed at the position indicated by the application during cursor movement.

For the multicolor cursor, these attributes are:

Multicolor cursor pattern
Sets the cursor shape and consists of a bit array defined by the application. The minimum size is 1 and the maximum size is 32 for both width and height.

Multicolor cursor foreground color
Sets the cursor foreground color and provides an index into the VLT.

Multicolor cursor background color
Sets the cursor background color and provides an index into the VLT.

Multicolor cursor origin
Sets the point in the cursor pattern that is placed at the position indicated by the application during cursor movement.

Multicolor cursor mask
Sets the mask to be applied to the multicolor cursor pattern, using a bit array defined by the application. The mask size must match the size of the multicolor cursor pattern.

Multicolor cursor logical operation
Defines how the GSL combines the pixels of the multicolor cursor with the current contents of the frame buffer.

At GSL initialization, some of the attributes receive default values. The attributes and their default values are listed in the following table:

Figure 6-2. Default Attribute Values

Attribute	**Default Value**
Color map	Device dependent
Plane mask	All planes enabled
Logical operation	3 (replace)
Line style	Solid
Line color (index)	7 (white)
Font	Device dependent

Code page	P0
Baseline direction	0 (left to right)
Text color (index)	7 (white)
Fill color (index)	7 (white)
Fill pattern	0 (solid)
Cursor pattern	Undefined
Cursor color	Undefined
Cursor origin	Undefined
Multicolor cursor pattern	Undefined
Multicolor cursor foreground color	Undefined
Multicolor cursor background color	Undefined
Multicolor cursor origin	Undefined
Multicolor cursor mask	Undefined
Multicolor cursor logical operation	Undefined
Xtext foreground color	Undefined
Xtext background color	Undefined
Xtext logical operation	Undefined
Xtext clip box	Undefined

2.6.2.4 Cursor Operations

The GSL provides one cursor for applications. The GSL cursor is non-destructive; the contents of the display adapter frame buffer remain intact when the cursor is already visible and subsequently moved or made invisible. This is achieved in a device-dependent manner. The GSL uses any hardware cursor support available.

The application totally controls the placement and visibility of the cursor. The GSL input functions do not provide cursor movement, and the GSL output routines do not check whether they are drawing over the cursor and do not automatically erase and restore the cursor or check for interference. It is therefore possible for the output routines to overwrite the cursor. When the cursor is moved on a display without hardware cursor support, any primitives that overwrite the cursor will themselves be overwritten when the area at the previous cursor position is restored. The application should erase and redraw the cursor as appropriate to avoid conflict.

2.6.2.5 Coordinate Clipping and Transformation

For simplicity and optimized performance, the base GSL does not perform general clipping or transformation on coordinates. Most of the output functions accept coordinates in the first quadrant (0,0 is the lower left corner) and convert them as necessary to the target quadrant required for the frame buffer of the specific device.

The coordinate system is device dependent, and any point outside the frame buffer range can result in a write to any address on the I/O bus. For this reason, the GSL checks coordinates sent as parameters and also coordinates generated internally against the frame buffer boundaries.

Note: Except for the **gsxtat** and **gsxtxt** routines, any coordinate outside the frame buffer is invalid and produces an invalid status return. (See the following explanation of xtext.)

The display results depend on the function invoked. Coordinates outside the frame buffer are handled as follows:

Lines The GSL checks the input coordinates as it draws the lines. Thus, for polylines, multilines, arcs, and ellipses, the part of the sequence up to the invalid coordinate results in lines on the display. The functions return an error code upon encountering an invalid coordinate, drawing no further lines.

Text If the start point of the text string is invalid, the GSL returns immediately with an error code. Invalid internal coordinates may be generated if the start point is valid but part of the text string overflows the frame buffer. If the application places the baseline such that the characters must be clipped vertically (for example, the top half of the string is out of the frame buffer), the GSL writes none of the characters in the string. If the application places the baseline such that a character in the string overflows the frame buffer, that entire character and the rest of the string is truncated.

Xtext The GSL provides clipping of this text style through the xtext clip box attribute. Coordinates outside the frame buffer are valid and will be clipped.

Filled Areas The GSL checks the coordinates of filled areas before it writes to the frame buffer. It returns an invalid return code for any invalid coordinate found before writing to the frame buffer.

Cursor The application may place the cursor origin anywhere within the cursor pattern. If the single-color cursor origin is placed so that any part of the cursor falls outside of the frame buffer, an error code is returned and the cursor is not moved. For the multicolor cursor, the application can place the origin anywhere within the cursor pattern. If the multicolor cursor origin is placed so that any part of the cursor falls outside the frame buffer, then

the part outside the frame buffer is clipped. If the origin of the multicolor cursor is placed outside the frame buffer, then an error code is returned and the cursor is not moved.

Pixel Block Transfer If any portion of the source or destination rectangle lies outside its pixmap, the GSL returns immediately with an error code.

2.6.3 Functional Categories of Subroutines

The base GSL is device dependent in that it provides some functions for some displays that it does not provide for other displays. It also does not scale, clip, or transform coordinates for any display. Coordinates passed to the base GSL functions are therefore device dependent, and limited to the device boundaries. The GSL does make device specific information available through query commands, letting an application perform appropriate clipping and transformation. It also indicates the logical operations supported by the hardware.

The base GSL is organized into several major function areas:

- Contro
- Outpu
- Servic
- Pixel Block Transfe
- Curso
- Attribut
- Inpu
- Query

The following sections provide an overview of the functions in each area.

Subtopics

2.6.3.1 Control

The base GSL provides functions to initialize and terminate the GSL and to coordinate direct application access to the display device. At initialization, the GSL sets up its required environment and establishes Monitor Mode on the application virtual terminal. Monitor Mode provides the GSL with direct access to the display adapter without interference from the virtual terminal subsystem. Monitor Mode operation also gives faster access to input event information. At termination, the GSL cleans up after itself and returns the application virtual terminal to KSR Mode.

These subroutines perform overall control operations of the GSL environment.

gsinit Initializes the GSL subroutines, establishes Monitor Mode on the application virtual terminal, and allows specification of application-supplied signal processing routines.

gslock Locks the virtual terminal so that the application can access the display adapter directly.

gsterm Terminates the GSL, returns the application virtual terminal to KSR Mode.

gsunlk Unlocks the virtual terminal, returning control to the GSL.

2.6.3.2 Output

The GSL output functions provide an application with capabilities to perform graphics operations on output devices. The output functions can be divided into these categories:

Drawing lines
The GSL provides functions to draw:

- A line between two points
- A series of lines connecting a sequence of points
- A series of lines connecting alternate pairs in a sequence of points.

GSL lines are a single pixel thick. Specific attributes allow lines of different colors and patterns.

Drawing polymarkers
The polymarker subroutine in the GSL lets a defined marker be drawn for a sequence of points. The definition of the marker includes specific attributes such as color, style, width, height, pattern, and origin. The pattern attribute is a raster image to be used as a marker. The origin attribute controls the placement of the polymarker pattern at the points specified by the polymarker subroutine.

Writing annotated text
The GSL provides a function to write a text string to the display adapter at a given starting position. Character placement is with a transparent background so that the GSL changes only the character shape (foreground), not the entire character box. Specific attributes allow text in different fonts, colors, code pages, and directions.

In addition, the GSL provides a text drawing facility that changes both the foreground and background of the frame buffer where the character box is placed. This facility, called ***xtext***, allows text in different fonts, foreground and background colors, and logical operations.

Writing geometric text
The GSL provides functions to write geometric text strings.

Drawing curves
The GSL provides functions to draw circles, arcs, and ellipses. These functions are used to achieve the best performance and quality possible so that your programs can realize the full capability of your display.

Filling areas
The GSL provides functions to draw filled rectangles and general polygons, circles, and ellipses. In addition, the GSL allows curves to be combined with polylines to fill complex shapes. See "gsbply" in topic 2.6.5, "gseply" in topic 2.6.19, and "gspcls" in topic 2.6.41. These functions allow applications to use the higher performance possible with rectangles. The GSL also provides a **color zero** function to clear the display to the background color. Specific attributes allow different colors and patterns. See "Attributes" in topic 2.6.2.3.

Each category of the output functions is governed by a set of attributes. Some attributes determine characteristics that are specific to the category, such as color or pattern. These attributes are common to all categories:

color table
Maps color names or values to the actual color on the display (see "Common Attributes" in topic 2.6.2.3.1).

plane mask
Determines which of the display adapter storage planes are modified by the output functions.

logical operation
Determines how the GSL combines the foreground or background color for each pixel that is produced by a primitive with the current color of the destination pixel in the frame refresh buffer.

These output subroutines write to the display adapter frame buffer, generally producing output on a display screen:

gsbply Begins a polygon.
gscarc Draws a circular arc of a specified radius between two points.
gscir Draws a circle.
gsclrs Clears the display screen, filling it with the background color.
gscrca Draws a circular arc between two angles.
gsdjply Draws a disjoint polyline, or a path of straight lines that are not connected.
gseara Draws an elliptical arc between two angles.
gsearc Draws an elliptical arc of specified axes and angle between two points.
gsell Draws an ellipse.
gseply Ends a polygon.
gsfci Fills a circle.
gsfell Fills an ellipse.
gsfrec Draws a filled rectangle.
gsfply Draws a filled polygon.
gsgtxt Displays a geometric text string, with NDC transformations supported.
gsline Draws a line between two points.
gsmult Draws a multiline, or a set of straight lines that connect alternate pairs of points in a sequence.
gspcls Closes a polygon.
gsplym Draws a polymarker, a marker (such as a dot or plus sign) at each of a specified set of points.
gspoly Draws a polyline, or a path of straight lines that connect a sequence of points.
gstext Displays a text string.
gsxtxt Displays a text string in the rtfont format.

2.6.3.3 Service

The GSL provides functions for defining a circular or elliptical arc. These functions convert circular or elliptical arc definitions into sets of vertices. The resulting set of vertices can be drawn, using the **gsline** subroutine, or combined with other polylines to draw or fill more complex shapes.

The attributes that can be used for drawing lines or filling areas apply here, including style, color, logical operation, pattern, and others.

Arcs are specified by beginning and ending points or beginning and ending angles, and follow the counterclockwise direction. If the beginning and ending points are identical, then the list of vertices corresponding to a full circle or ellipse is returned. This allows circles or ellipses to be treated as a special case of closed arcs. If off-axis, ellipsis angle is specified in degrees. There are four levels of precision for the conversion of an arc into a set of line segments.

These subroutines facilitate the drawing of circular and elliptical arcs.

gsccnv Converts a circle to a set of vertices (polyline).
gsecnv Converts an ellipse to a set of vertices (polyline).

2.6.3.4 Pixel Block Transfer

The GSL provides functions to move a rectangular block of pixels from either the display adapter frame buffer or storage to either the display adapter or storage. If the source rectangle or destination rectangle reside in a color display adapter frame buffer, this operation is affected by the plane mask attribute. If the destination rectangle resides in a color display adapter frame buffer, this operation is affected by the color map attribute.

These subroutines allow a program to:

- Save a block of pixels from the frame buffer
- Restore a block of pixels from the frame buffer
- Move a rectangular shape from system memory to adapter memory
- Move a rectangular shape from adapter memory to system memory
- Move a rectangular shape from one area in system memory to another
- Move a rectangular shape from one area of adapter memory to another
- Move a tile rectangle to any area of visible pixel memory

gsrrst	Restores a rectangular block.
gsrsav	Saves a rectangular block.
gsxblt	Moves a rectangular block from one location in memory or display adapter frame buffer to another location in memory or display adapter frame buffer.
gsxcnv	Converts pixel format data to and from plane format data.
gsxptr	Handles FORTRAN addressing of pixel map data.

2.6.3.5 Cursor

The GSL provides functions to draw and undraw a non-destructive cursor. The application is responsible for the placement and visibility of the cursor. The input functions do not provide for cursor movement, nor do output or pixel block transfer functions check whether they interfere with the cursor. Anything unintentionally placed over the cursor is modified when the cursor moves. The color map and plane mask attributes govern the cursor functions. Cursor pattern and color can be defined by attributes.

These are the cursor subroutines:

gsecur Erases the cursor and makes it invisible.
gsmcur Moves the cursor and makes it visible.

2.6.3.6 Attribute

The GSL provides functions to set the global attributes and all of the output category specific attributes. The GSL also provides functions to set attributes of some of the input devices.

These subroutines set attributes for both input and output operations:

gscatt Sets the single-color cursor attributes.
gscmap Sets the color map.
gsfatt Sets the fill attributes.
gsgtat Sets the attributes for the geometric text drawing operation, **gsgtxt**.
gslatt Sets the line attributes.
gslcat Sets the locator attributes.
gslpat Sets the LPFK indicators.
gslop Sets the logical operation used for drawing lines.
gsmask Sets the plane mask.
gsmatt Sets the attributes for the polymarker operation, **gsplym**.
gsmcat Sets the multicolor cursor attributes.
gspp Sets plotter pen speed as a percentage of the plotter maximum speed.
gstatt Sets the attributes for the text output operation, **gstext**.
gsulns Sets the user line pattern.
gsvgrn Sets the valuator granularity.
gsxtat Sets the attributes for drawing annotated text in the rtfont format, using the **gsxtxt** routine.

2.6.3.7 Input

An application using the GSL can receive input with the standard **read** system call or through a faster mechanism available through the virtual terminal. While the GSL allows an application to use the standard mechanism, it provides no input support for it.

The GSL accepts input from the following sources:

The keyboar
The locato

Input from these devices is viewed as discrete events, with input data associated with each event.

The GSL provides subroutines to enable or disable input from any device, and a subroutine that lets a program suspend execution until one of the enabled events occurs. The latter subroutine also parses the raw data generated by the virtual terminal and makes the parsed information available to the application.

The state established (enabled or disabled) remains in effect when the GSL terminates. Note that for these subroutines to work properly, a valid input ring buffer must have been specified to the **gsinit** subroutine.

gsdpik	Disables picking.
gsepik	Enables picking.
gsevds	Disables the reporting of input events.
gseven	Enables the reporting of input events from the keyboard or the locator.
	Enables the reporting of input events from the keyboard, locator, LPFK, or valuator.
gsevwt	Waits for an input event and parses the raw data.

2.6.3.8 Query

The GSL provides functions for applications to query the active display adapter characteristics, the currently active annotated or geometric text font, and some input device characteristics. Through query functions, an application can derive the information necessary to deal with any device dependencies. Note that **gsinit** must be invoked before calling any of the query subroutines.

These are subroutines that provide query functions:

gsqdsp	Returns device-specific information about the display adapter and display monitor.
gsqfnt	Returns information about the current annotated text font.
gsqgtx	Returns information about the current geometric text font.
gsqloc	Returns device-specific information about the locator.

2.6.4 Writing GSL Application Programs

The following sections contain information that will help you take advantage of the capabilities of the GSL when writing application programs. "Displays" describes the display support provided by the GSL; "Printers and Plotters" explains available printer and plotter support; and "Using the GSL Libraries" contains instructions on using libraries and setting up the installation of your application program.

Subtopics
2.6.4.1 Displays
2.6.4.2 Printers and Plotters
2.6.4.3 Using the GSL Libraries

2.6.4.1 Displays

The AIX PS/2 Graphic Support Library provides support for VGA graphics with 640 x 480 resolution displaying 16 of 256,000 colors or 16 of 64 possible levels of gray. The following monitors are supported on the VGA:

8503	12 inch	monochrome
8507	15 inch	monochrome
8604	19 inch	monochrome
8512	14 inch	color
8513	12 inch	color
8514	16 inch	color

The AIX PS/2 Graphic Support Library provides support for the 8514A graphics adapter with 1024 x 768 resolution displaying 256 of 256,000 colors or 64 of 64 possible levels of gray. The following displays are supported on the 8514A adapter:

8507	15 inch	monochrome
8604	19 inch	monochrome
8514	16 inch	color

The following displays connected to the 8514A adapter are supported in the VGA 640 x 480 resolution passthru mode with 16 of 256,000 colors or 16 of 64 levels of gray:

8503	12 inch	monochrome
8512	14 inch	color
8513	12 inch	color

The AIX PS/2 Graphic Support Library provides support in a mode according to display and display adapter information passed to the AIX Operating System by the system administrator at boot time. If a display is connected to the 8514A and is capable of running in either 1024 x 768 or 640 x 480 resolution, then the VGA passthru mode with 640 x 480 picture element resolution may be selected by setting and exporting the following environmental variable:

GSL_RESOLUTION Set to 640x480. This is an AIX PS/2 Release 1 specific option. It may not be supported in future releases.

The GSL supports one or more of the following input devices in an application:

Keyboar
Mous

At least one input device is always available; the virtual terminal subsystem determines that, at minimum, keyboard input is accepted.

2.6.4.2 Printers and Plotters

Certain information about the device must be defined using AIX environment variables. If you enter the definitions at the shell command line, then they remain in effect only for the current login session. If you want these definitions to remain in effect for future login sessions, add them to the **.profile** file in your home directory. To define this information permanently for all users, add it to the **/etc/profile** file. See the **sh** command in *AIX Operating System Commands Reference* for more information about AIX environment variables, which are also called ***shell variables***.

1. Define the path to the AIX PS/2 hardcopy device drivers:

   ```
   VDIPATH=/usr/lpp/vdi/drivers
   export VDIPATH
   ```

2. Define a logical identifier for the device as an environment variable, and set its value to indicate the type of printer or plotter device:

   ```
   devname=cgixxxx
   export devname
   ```

   The name you use in place of **devname** can be any sequence of up to 11 alphanumeric characters. This is the name that you specify in the **fildes** parameter of the **gsinit** subroutine.

   The value of the environment variable, **cgixxxx**, must be one of the following:

   cgipro IBM 4201, 4202 Proprinters
   cgiplot IBM 7371, 7372, 7374, 7375-1, 7375-2, 6180, 6182, 6184, 6186-1, or 6186-2 Plotter

3. You can specify a printer or plotter as **cgixxxx** in step 2; **cgixxxx** must be associated with an AIX special file:

   ```
   cgixxxx=/dev/yyyy
   export cgixxxx
   ```

   If you do not need output from a specific printer device, you can pipe the output to a printer queue:

   ```
   cgixxxx="| print -plot [queue]"
   export cgixxxx
   ```

   Note: You can only pipe output to a queue for printer devices, not for plotters.

   Another alternative is to assign **cgixxxx** to a regular file for later use.

4. If you are using an IBM 420X Proprinter, you may set and **export** the following environment variables:

   ORIENTATION Set to either **LANDSCAPE** or **PORTRAIT**, indicating horizontal or vertical page orientation, respectively. ***Landscape orientation*** rotates the image 90 degrees so that the horizontal axis of the image goes down the

length of the page. If not defined, and **PAPER** is set to **WIDE**, the default is **LANDSCAPE** mode. For **PAPER** set to **NARROW**, the default orientation is **PORTRAIT** mode.

PAPER Set to either **WIDE** or **NARROW**, indicating the width of the paper loaded in the printer. The default setting for **PAPER** is **NARROW**.

PRINTERTYPE Set to **I**, **II**, or **XL**, indicating the type of Proprinter. The default setting is II.

5. If you are using an IBM plotter, you may set and **export** the following environment variables:

ORIENTATION As for the printers, set to either **LANDSCAPE** or **PORTRAIT**. For plotters, the default setting is **LANDSCAPE**.

MESSAGEPORT Set to the desired destination for plotter prompts. The default is the current controlling terminal. For example:

```
MESSAGEPORT=/dev/lp1
export MESSAGEPORT
```

would send all plotter messages to **lp1**. Unless the message port is set to **/dev/lp**, the GSL hardcopy drivers will set up a **lock** file in the directory **/usr/spool/uucp** for the specified **lp** or virtual terminal.

6. Plotters should be set up using the following configuration:

BAUD 9600

STOP BITS 1

PARITY None

7. **Plotter Notes**

IBM 6184

The 6184 plotter's default coordinate system is rotated 180 degrees. This means that all plots are plotted upside down.

IBM 6186

If pen sort is **ON**, unexpected results may occur because the order in which vectors are drawn is not necessarily the order in which the plotter received them. The unknown pen position may affect subsequent output instructions.

Examples

The following example defines **GRAPHDEV** as the logical name of an IBM 6182 plotter that is configured as **/dev/lp1**. This configuration specifies ***portrait orientation*** (output frame vertical dimensions are greater than horizontal). Plotter prompts are directed to **/dev/lp1**.

```
VDIPATH=/usr/lpp/vdi/drivers
ODEVICE=cgiplot
cgiplot=/dev/lp1
MESSAGEPORT=/dev/lp1
ORIENTATION=PORTRAIT
export VDIPATH ODEVICE cgiplot ORIENTATION MESSAGEPORT
```

The next example defines **ODEVICE** as the logical name of an IBM 4202 printer that is already configured as the device that serves printer queue **lp2**. This configuration specifies ***landscape orientation*** (output frame horizontal dimensions are greater than vertical) and wide paper.

```
VDIPATH=/usr/lpp/vdi/drivers
ODEVICE=cgipro
cgipro="| print -plot lp2"
PAPER=WIDE
ORIENTATION=LANDSCAPE
export VDIPATH ODEVICE cgipro ORIENTATION PAPER
```

Note: The GSL is available for printer and plotter devices except for "Pixel Block Transfer", "Cursor", and "Xtext" subroutines.

2.6.4.3 Using the GSL Libraries

Two versions of the GSL subroutine library are provided in **/usr/lib**:

An unshared library, named **libgsl.a**

A shared library, named **libogsl.a**, and its corresponding text image file, **ogsl.txt**.

IBM strongly recommends that you use the shared GSL library, since doing so can save virtual memory and disk storage space. For more information about shared libraries, see "Shared Libraries" in *AIX PS/2 Programming Tools and Interfaces* and the **shlibrpt** and **shlib2** commands for the RT and PS/2, respectively, in *AIX Operating System Commands Reference*.

A GSL application should be relinked any time that the GSL libraries are updated. If, for example, support for a new type of display is added to the libraries and the application is not relinked, then the application cannot access the new display and may produce unpredictable results.

You can relink GSL applications yourself, but the libraries can be updated without your knowledge. Therefore, IBM recommends a facility that automatically relinks all GSL applications when the libraries are changed. To use this facility, your application should follow these conventions:

Be provided as object files, **not** as prelinked executables.

Provide an **sh** shell script named **lpp.linkgsl** that links the object files into the final executable form.

The **lpp.linkgsl** shell script must be located in a directory named **/usr/lpp/pgm_name**, where **pgm_name** is a name that uniquely identifies your application program. Whenever the GSL libraries are changed, such as for the addition of new display support, each of the **/usr/lpp/pgm_name** directories is searched for an **lpp.linkgsl** file. Each **lpp.linkgsl** is executed, relinking the GSL applications with the updated subroutine library.

Subtopics

2.6.4.3.1 Notes on the lpp.linkgsl Shell Script

1. If you are writing an application program to be installed on more than one system, IBM strongly recommends that you package your application to be installed from diskette with the standard AIX conventions of the **installp** command. Your **instal** procedure should run **lpp.linkgsl** to link the application with the GSL initially. You can find instructions for creating a diskette in the proper format under "Installing and Updating a Program" in *AIX PS/2 Programming Tools and Interfaces*.

2. IBM recommends that you process the object files of your application with **ld -r** to link them into one final object file that can still be linked with libraries. You can do this before creating the **installp** diskette. The following example combines all of the **my*.o** object files into a final object file named **mypgm.o**, which can be shipped on an **installp** diskette:

   ```
   ld -r -o mypgm.o my*.o
   ```

3. IBM recommends that you store the object files for your application in a directory named **/usr/lpp/pgm_name/bin**. This prevents the **installp** command from deleting them after installation.

4. Your **lpp.linkgsl** script should use **libgsl.a** if **libogsl.a** is not available.

2.6.4.3.2 Example lpp.linkgsl Shell Script

A typical **lpp.linkgsl** script might look like this:

```
  # The current directory has already been set to /usr/lpp/mypgm

  if [ -r /usr/lib/libogsl.a ]
  then
    GSLLIB=ogsl
  else
    # May be running on an older version of AIX that doesn't
    #   have the shared library.  Use the unshared version.
    GSLLIB=gsl
  fi

  # bin/mypgm.o was previously linked with:
  #   ld -r -o mypgm.o my*.o

  cc -o mypgm bin/mypgm.o -l$GSLLIB
  rc = $?
  if [ $rc -ne 0 ]
  then
    echo "Failed trying to relink mypgm." >&2
  else
    # Move executable to /usr/bin, where users can run it
    cp mypgm /usr/bin/mypgm
    rm -f mypgm
  fi
```

2.6.4.3.3 GSL Hardcopy Error Codes

The following is a list of the errors that can be generated and a brief description of the error:

-2000 Illegal handle

Cause: An incorrect device handle was specified.

-2001 Unknown driver file

Cause: The device driver file does not exist.

-2002 No driver file

Cause: The name of the logical device was not found.

-2003 Cannot start device driver

Cause: The **fork** command failed to start the graphics device driver as a new process.

-2004 No more device drivers can be opened

Cause: The device driver cannot start because the maximum number of device drivers are already opened.

Each application is limited to having no more than eight graphics devices open simultaneously.

-2005 Cannot create semaphore

Cause: The device driver cannot start because a system semaphore cannot be acquired.

-2006 Cannot create shared memory area

Cause: The device driver cannot start because a system-shared memory area cannot be acquired.

-2007 Cannot attach memory area

Cause: The device driver cannot start because a system-shared memory area cannot be attached to the device driver process.

-2019 Device busy error

Cause: The requested channel's physical device is in use and, therefore, not available.

This error is generated when a lock file has been created for the device you are attempting to use. A lock file is created each time you perform graphics to a device that is not assigned to the logical device **/dev/tty**. The naming of the lock file has the form:

```
LCK..ttynn
```

where **nn** is the identification of the logical device being requested.

-2020 Open device error

Cause: The physical device associated with the requested channel could not be opened.

-2021 Create lockfile error

Cause: Unable to create the lock file for the requested logical device.

-2022 Unable to create process

Cause: The attempt to create a process to associate with the requested channel failed.

-2023 File creation error

Cause: Attempt to create a file to be used for I/O redirection failed.

-2024 Invalid device assignment

Cause: The requested channel has an invalid device assignment.

-2025 Could not open font file

Cause: The Font Manager could not open the requested font file.

-2026 Font does not exist

Cause: The requested font index is out of range or the font does not exist.

-2027 Font file missing

Cause: The Font Manager could not find the requested font file. This can indicate that the font database file is out of date.

-2028 Missing font manager database file

Cause: The Font Manager could not find the database file **fontlist.dat**.

-2029 Read font file error

Cause: The Font Manager encountered an error while reading the font file.

-2977 CGI already loaded

Cause: This error is returned from Load CGI and should be ignored. It is there for compatibility with DOS applications.

-3000 Device driver error

Cause: An error occurred when calling the device driver.

-3001 Device driver invalid range error

Cause: A parameter that was passed from the application was out of the valid range for the operation.

-3002 Handle in use error

Cause: The bitmap handle is in use and cannot be deleted.

-3004 Incompatibility error

Cause: The device driver and the CGI controller are incompatible.

-3005 Too big bitmap error

Cause: The requested bitmap is too large.

-3008 Device driver could not malloc

Cause: There is insufficient memory to create the intended bitmap.

-3090 Invalid page error

Cause: The physical page is invalid.

-3095 Device not in cursor mode

Cause: The device is not in cursor text mode.

-5000 Device driver not capable error

Cause: The specified device is not capable of performing the requested function.

2.6.5 gsbply

Purpose
Defines the beginning of an area to fill.

C Syntax

int gsbply_ ( )

FORTRAN Syntax

INTEGER function gsbply

Pascal Syntax

FUNCTION gsbply_ : INTEGER [**PUBLIC];**

Description

The **gsbply** subroutine defines the beginning of a two-dimensional shape or set of shapes to be filled.

The following output routines are valid between a **gsbply** call and a **gseply** call:

Draw polyline	**gspoly**)
Draw circle	**gscir**)
Draw ellipse	**gsell**)
Draw circular arc	**gscarc** or **gscrca**)
Draw elliptical arc	**gseara** or **gsearc**)

Note: Any other subroutines used before the **gseply** subroutine is called do not become part of the shape or set of shapes to be filled, and can produce unpredictable results.

Before the fill occurs, the shapes drawn by each routine called between **gsbply** and **gseply** are connected. The first point of each shape is linked to the last point of the previous shape, and the last point of the last shape is linked to the first point of the first shape. The shapes may overlap to any degree but must share at least one common point between adjacent shapes.

Processing of the **SIGRETRACT** signal is postponed until the **gseply** subroutine, end of area to fill, is called.

The relevant attributes are:

Color ma
Plane mas
Fill color inde
Fill styl
Logical operation

Return Value

GS_SUCC Successful.
GS_USUC Unsuccessful.

Related Information
In this book: "gseply" in topic 2.6.19 and "gspcls" in topic 2.6.41.

2.6.6 gscarc

Purpose
Draws a circular arc between two points.

C Syntax

int gscarc_ (cx, cy, cr, bx, by, ex, ey)

int *cx, *cy, *cr, *bx, *by, *ex, *ey;

FORTRAN Syntax

INTEGER function gscarc (cx, cy, cr, bx, by, ex, ey)

INTEGER cx, cy, cr, bx, by, ex, ey

Pascal Syntax

FUNCTION gscarc_ (

VAR cx, cy, cr, bx, by, ex, ey : INTEGER
): INTEGER [**PUBLIC];**

Description

The **gscarc** subroutine draws a counterclockwise circular arc of the specified radius from the beginning point to the ending point. The radius is expressed in number of pixels.

The relevant attributes are:

- Color ma
- Plane mas
- Line color inde
- Line styl
- Logical operation

Subtopics
2.6.6.1 Parameters

2.6.6.1 Parameters

cx, cy Define the coordinates of the center of the circle.

For displays, the center is restricted to -2048 to 2048.

For printers and plotters, the center is restricted to screen coordinates.

cr Defines the radius of the circle.

bx, by Define the coordinates of the beginning point on the circle.

ex, ey Define the coordinates of the ending point on the circle.

If the beginning and ending points are identical, a full circle is drawn.

Note that the application must control the accuracy of the end points (**bx, by** and **ex, ey**) when drawing circular arcs. If the start point of the arc and end point of the arc lie within one pixel of the true circle, the arc will be drawn successfully. Other values can cause the subroutine to fail. If the **gscarc** subroutine fails because of an inaccurate starting point, **GS_ASTR** is returned, while for an inaccurate ending point, **GS_AEND** is returned.

Return Value

GS_SUCC Successful.
GS_CORD Invalid coordinate.
GS_RDUS Invalid radius for circles.
GS_INAC Virtual terminal inactive.
GS_AEND Invalid end point.
GS_ASTR Invalid start point.

2.6.7 gscatt

Purpose
Sets the attributes of the single-color cursor.

C Syntax

```
int gscatt_ (color, width, height, pattern, 0x, 0y)

int *color, *width, *height, *pattern, *0x, *0y;
```

FORTRAN Syntax

```
INTEGER function gscatt (color, width, height, pattern, 0x, 0y)

INTEGER color, width, height, pattern, 0x, 0y
```

Pascal Syntax

```
FUNCTION gscatt_ (

VAR color, width, height: INTEGER;
pattern: ARRAY [1..k] of INTEGER;
0x, 0y: INTEGER
): INTEGER [PUBLIC];
```

Description
The **gscatt** subroutine defines the single-color cursor for the GSL. The **gscmap** subroutine must initialize the color map before **gscatt** can be called.

Only one cursor, either the single-color cursor or the multicolor cursor, can be active in the GSL at any one time. The **gscatt** subroutine forces all subsequent calls to the **gsmcur** and **gsecur** subroutines to operate on the single-color version of the cursor. To change from the multicolor cursor to the single-color cursor, erase the cursor with **gsecur**, then call the **gscatt** subroutine.

Subtopics
2.6.7.1 Parameters

2.6.7.1 Parameters

color Refers to an entry in the color map. If the index value is -1, the attribute is unchanged.

width, height Define, in pixels, the width and height of the bit pattern to be used as the cursor. If **width** or **height** equals -1, then the pattern remains unchanged.

pattern Defines the image used as a cursor. The ceiling (**width**/32) indicates the number of words per row and **height** indicates the number of rows. The cursor data must be supplied in row (scan line) major order. If **width** implies partial use of a word, the rest of the word is unused. To fully define the cursor pattern, **pattern** should be (ceiling¦**height**) words in length.

0x, 0y Indicate the origin of the cursor relative to the lower leftmost corner (0, 0) of the cursor pattern. The origin must be placed within the cursor pattern: **0x** < **width** and **0y** < **height**. The origin of the cursor is placed at the position indicated, when the application moves the cursor using the **gsmcur** subroutine. If **x** equals -1, then the origin remains unchanged.

The maximum size of the cursor is device dependent and can be determined by using the **gsqdsp** subroutine.

You cannot change the cursor attributes while the cursor is visible.

There is no default cursor defined, so all cursor parameters must be set before the cursor is displayed.

For Pascal, the application must declare the array passed as being fixed length and declare the routine as accepting arrays of that length. The **k** in the routine declaration must be a constant.

Return Value

GS_SUCC Successful.
GS_COLI Invalid color index.
GS_CURS Cursor size invalid.
GS_CURO Cursor origin invalid.
GS_CURV Cursor visible.

Related Information
In this book: "gsecur" in topic 2.6.17, "gsmcat" in topic 2.6.38, and "gsmcur" in topic 2.6.39.

2.6.8 gsccnv

Purpose
Converts a circular arc or full circle into a polyline.

C Syntax

```
int gsccnv_ (cx, cy, cr, bx, by, ex, ey, len, x, y, pre)

int *cx, *cy, *cr, *bx, *by, *ex, *ey, *len, *x, *y, *pre;
```

FORTRAN Syntax

```
INTEGER function gsccnv (cx, cy, cr, bx, by, ex, ey, len, x, y, pre)

INTEGER cx, cy, cr, bx, by, ex, ey, len, x(*), y(*), pre
```

Pascal Syntax

```
FUNCTION gsccnv_ (

VAR cx, cy, cr, bx, by, ex, ey, len: INTEGER;
VAR x, y: ARRAY [1..k] of INTEGER;
VAR pre: INTEGER
): INTEGER [PUBLIC];
```

Description
The **gsccnv** subroutine converts a counterclockwise circular arc definition into an array of vertices. The list of vertices can then be used to draw a circular arc with the **gspoly** subroutine or to fill a circular arc with the **gsfply** subroutine. In general, it can be concatenated with other list(s) of vertices to draw or fill more complex shapes, such as chord arcs, pie arcs, and rectangles with rounded corners.

When beginning and ending points are identical, the list of vertices contains the full circle, which can then be drawn or filled.

Subtopics
2.6.8.1 Parameters

2.6.8.1 Parameters

cx, cy Define the coordinates of the center of the circle.

cr Defines the radius of the circle, which must not be equal to 0.

If **cr** is negative, it is automatically converted to a positive value for use by the subroutine.

bx, by Define the coordinates of the beginning point of the arc.

ex, ey Define the coordinates of the ending point of the arc.

len Defines the number of points in the coordinate **x** and **y** arrays. It must be numerically at least one greater than the value contained in the precision parameter, but not less than 65.

x, y Define, as coordinate arrays, the vertices that represent the circular shape when drawn or filled.

pre Defines precision level, which specifies the maximum number of line segments that can be generated for a full circle. The number of line segments actually generated depends on the size of the circle.

There are four levels of precision that can be requested:

64 (65 vertices)

128 (129 vertices)

256 (257 vertices)

512 (513 vertices).

Therefore, **len** = **pre** + 1.

All other precision values are reserved and must not be used, as their results are unpredictable. The default value for **pre** is 64.

The subroutine allows ample leniency toward the accuracy of the specification of the beginning and ending points. The arc of the specified radius will always start and end exactly at the specified points.

If the beginning and ending points are identical, a full circle of the specified radius is generated.

When the subroutine is invoked, the length parameter must contain the maximum number of entries in the **x** and **y** arrays. If erroneous conditions arise, **len** is set to 0. Under normal conditions, **len** specifies the number of vertices returned by the subroutine in the **x** and **y** arrays.

For Pascal, the application must declare the arrays passed as being fixed length and declare the routine as accepting arrays of that length; the **k** in the routine declaration must be a constant.

Return Value

GS_SUCC Successful.
GS_CORD Invalid coordinate.
GS_NCOR Invalid number of coordinates.

2.6.9 gscir

Purpose
Draws a circle.

C Syntax

int gscir_ (cx, cy, cr)

int *cx, *cy, *cr;

FORTRAN Syntax

INTEGER function gscir (cx, cy, cr)

INTEGER cx, cy, cr

Pascal Syntax

FUNCTION gscir_ (

VAR cx, cy, cr: INTEGER
): INTEGER [**PUBLIC];**

Description

The **gscir** subroutine draws a circle of the specified radius. The radius is expressed in number of pixels.

The relevant attributes are:

- Color ma
- Plane mas
- Line color inde
- Line styl
- Logical operation

Subtopics
2.6.9.1 Parameters

2.6.9.1 Parameters

cx, cy Define the coordinates of the center of the circle.

cr Defines the radius of the circle.

If the radius is 0, a single point is drawn at the center.

Return Value

GS_SUCC Successful.
GS_CORD Invalid coordinate.
GS_RDUS Invalid radius specification.
GS_INAC Virtual terminal inactive.

2.6.10 gsclrs

Purpose
Clears the screen, filling it with the background color.

C Syntax

int gsclrs_ ( )

FORTRAN Syntax

INTEGER function gsclrs

Pascal Syntax

FUNCTION gsclrs_: INTEGER [**PUBLIC];**

Description
The **gsclrs** subroutine fills the frame buffer with the background color (color index zero).

The relevant attributes are:

- Color map
- Plane mask

For printers, the **gsclrs** subroutine forces pending graphics to be printed, advances the paper to a new page, and purges the print buffer.

For plotters, the **gsclrs** subroutine forces pending graphics to be **displayed**, and issues a prompt to the active screen (console) requesting that the paper be changed.

Return Value

GS_SUCC Successful.
GS_INAC Virtual terminal inactive.

2.6.11 gscmap

Purpose
Specifies the color mapping.

C Syntax

int gscmap_ (number, red, green, blue)

int *number, *red, *green, *blue;

FORTRAN Syntax

INTEGER function gscmap (number, red, green, blue)

INTEGER number, red (*), green (*), blue (*)

Pascal Syntax

FUNCTION gscmap_ (

VAR number INTEGER;
VAR red, green, blue: ARRAY [**0..k**] **of INTEGER**
): INTEGER [**PUBLIC**];

Description
The **gscmap** subroutine specifies the mapping between the color index attribute and the color it produces on the display.

The default color table mapping for the first 16 colors is the same as the default color map attributes in KSR mode. The remaining color values are initialized in a hardware dependent manner.

Subtopics
2.6.11.1 Parameters

2.6.11.1 Parameters

number Indicates how many colors the input intensity arrays contain.

red, green, blue
Define arrays that contain the intensity levels of the corresponding color. Each entry in an array specifies the intensity value for the corresponding color index.

The value in each entry for the **red**, **green**, and **blue** intensity arrays is between 0x0000, representing zero intensity, and 0x3FFF, representing full intensity. The following additional increments of intensity are possible, depending on the adapter hardware in use:

0x2000 1/2 intensity
0x1000 1/4 intensity
0x0800 1/8 intensity
0x0400 1/16 intensity
0x0200 1/32 intensity
0x0100 1/64 intensity.

Combinations of these values can be used to create intermediate levels of intensity. For example, **0x0C00** gives 3/16 intensity, while **0x3000** gives 3/4 intensity.

The actual number of bits from bit 13 to bit 0 that affect the color on the display is dependent on the number of bits in the digital-to-analog converter of the adapter hardware in use. This size information is available by using the **gsqdsp** subroutine.

An application cannot change a single arbitrary color entry in the color map (or the VLT). It must change all the entries for all the colors up to and including the desired entry.

For Pascal, the application must declare the arrays passed as being fixed length and declare the routine as accepting arrays of that length; the **k** in the routine declaration must be a constant.

Return Value

GS_SUCC Successful.
GS_TABL Invalid table length.
GS_INAC Virtual terminal inactive.

2.6.12 gscrca

Purpose
Draws a circular arc between two angles.

C Syntax

int gscrca_ (cx, cy, cr, ba, ea)

int *cx, *cy, *cr, *ba, *ea;

FORTRAN Syntax

INTEGER function gscrca (cx, cy, cr, ba, ea)

INTEGER cx, cy, cr, ba, ea

Pascal Syntax

FUNCTION gscrca_ (

VAR cx, cy, cr, ba, ea : INTEGER
): INTEGER [**PUBLIC];**

Description

The **gscrca** subroutine draws a counterclockwise circular arc of the specified radius from the beginning point as defined by an angle specification to the ending point as defined by an angle specification.

The angle specifications are given in tenths of degrees, from 0 to 3600. Values outside this range cause the **gscrca** subroutine to fail.

The relevant attributes are:

    Color ma
    Plane mas
    Line color inde
    Line styl
    Logical operation

Subtopics
2.6.12.1 Parameters

2.6.12.1 Parameters

cx, cy Define the coordinates of the center of the circle.

For displays, the center is restricted to -2048 to 2048.

For printers and plotters, the center is restricted to screen coordinates.

cr Defines the radius of the circle in device coordinates.

ba Defines the start point of the circular arc as an angle in tenths of degrees, from 0 to 3600.

ea Defines the end point of the circular arc as an angle in tenths of degrees, from 0 to 3600.

If the beginning and ending angles are identical, a full circle is drawn.

Return Value

GS_SUCC Successful.
GS_ANGL Invalid angle.
GS_RDUS Invalid radius specification.
GS_CORD Invalid coordinate.
GS_INAC Virtual terminal inactive.

2.6.13 gsdjply

Purpose
Draws a polyline, a set of lines that connects a sequence of points.

C Syntax

```
int gsdjply_ (polylines, points, x, y)

int *polylines, *points, *x, *y;
```

FORTRAN Syntax

```
INTEGER function gsdjply (polylines, points, x, y)

INTEGER polylines, points (*), x (*), y (*)
```

Pascal Syntax

```
FUNCTION gsdjply_ (

VAR polylines: INTEGER;
VAR points: ARRAY [1..k] of INTEGER;
VAR x, y: ARRAY [1..l] of INTEGER
): INTEGER [PUBLIC];
```

Description
The **gsdjply** subroutine draws a series of polylines as a set of lines, as defined by the current relevant attributes.

The relevant attributes are:

- Color ma
- Plane mas
- Line color inde
- Line styl
- Logical operation

Subtopics
2.6.13.1 Parameters

2.6.13.1 Parameters

polylines Defines the number of polylines to draw. This value must be = 1.

points Defines, as an array, the number of points in each polyline. These values must be = 2.

x, y Define, as an array, the points for line drawing.

For Pascal, the application must declare the arrays passed as being fixed length and declare the routine as accepting arrays of that length. The **k** and **l** in the routine declaration must be constants.

Return Value

GS_SUCC Successful.
GS_CORD Invalid coordinate.
GS_NCOR Invalid number of coordinates.
GS_INAC Virtual terminal inactive.

2.6.14 gseara

Purpose
Draws an elliptical arc between two angles.

C Syntax

int gseara_ (cx, cy, ma, mi, ang, sa, ea)

int *cx, *cy, *ma, *mi, *ang, *sa, *ea;

FORTRAN Syntax

INTEGER function gseara (cx, cy, ma, mi, ang, sa, ea)

INTEGER cx, cy, ma, mi, ang, sa, ea

Pascal Syntax

FUNCTION gseara_ (

VAR cx, cy, ma, mi, ang, sa, ea : INTEGER
): INTEGER [**PUBLIC];**

Description

The **gseara** subroutine draws a counterclockwise elliptical arc of the specified axes and angle from the beginning point defined by an angle specification to the ending point defined by an angle specification. The axes are expressed in number of pixels.

The angle specifications are given in tenths of degrees, from 0 to 3600. Values outside this range cause the **gseara** subroutine to fail.

The relevant attributes are:

Color ma
Plane mas
Line color inde
Line styl
Logical operation

Subtopics
2.6.14.1 Parameters

2.6.14.1 Parameters

cx, cy Define the coordinates of the center of the ellipse.

For displays, the center is restricted to -2048 to 2048.

For printers and plotters, the center is restricted to screen coordinates.

ma, mi Define half of the nonzero major and minor axes of the ellipse.

ang Defines the angle between the major axis and the x-axis. If **ang** is 0, the major axis is on the x-axis and the minor axis is on the y-axis. The angle is expressed in tenths of degrees, from 0 to 3600.

sa Defines the angle of the starting point of the elliptical arc, measured counterclockwise from the major axis. The angle is expressed in tenths of degrees, from 0 to 3600.

ea Defines the angle of the ending point of the elliptical arc, measured counterclockwise from the major axis. The angle is expressed in tenths of degrees, from 0 to 3600.

If the beginning and ending points are identical, a full ellipse is drawn.

Return Value

GS_SUCC Successful.
GS_CORD Invalid coordinate.
GS_ELMM Invalid major or minor axis.
GS_INAC Virtual terminal inactive.
GS_ANGL Invalid angle.
GS_NMEM No memory available.

2.6.15 gsearc

Purpose
Draws an elliptical arc between two points.

C Syntax

int gsearc_ (cx, cy, ma, mi, ang, bx, by, ex, ey, rot)

int *cx, *cy, *ma, *mi, *ang, *bx, *by, *ex, *ey, *rot;

FORTRAN Syntax

INTEGER function gsearc (cx, cy, ma, mi, ang, bx, by, ex, ey, rot)

INTEGER cx, cy, ma, mi, ang, bx, by, ex, ey, rot

Pascal Syntax

FUNCTION gsearc_ (

VAR cx, cy, ma, mi, ang, bx, by, ex, ey, rot : INTEGER
): INTEGER [**PUBLIC];**

Description

The **gsearc** subroutine draws a counterclockwise elliptical arc of the specified axes and angle from the beginning point to the ending point. The axes are expressed in number of pixels.

The angle specifications are given in tenths of degrees, from 0 to 3600. Values outside this range cause the **gsearc** subroutine to fail.

The relevant attributes are:

- Color ma
- Plane mas
- Line color inde
- Line styl
- Logical operation

Subtopics
2.6.15.1 Parameters

2.6.15.1 Parameters

cx, cy Define the coordinates of the center of the ellipse.

For displays, the center is restricted to -2048 to 2048.

For printers and plotters, the center is restricted to device coordinates.

ma, mi Define half of the nonzero major and minor axes of the ellipse.

ang Defines the angle between the major axis and the x-axis. If **ang** is 0, the major axis is on the x-axis and the minor axis is on the y-axis. The angle is expressed in tenths of degrees, from 0 to 3600.

bx, by Define the coordinates of the beginning point on the ellipse.

ex, ey Define the coordinates of the ending point on the ellipse.

rot Specifies whether the application must perform rotational transformation. Possible setting are:

0 The coordinates of the beginning and ending points passed by the application correspond to an arc of an orthogonal ellipse. No rotational transformation is performed, thus improving performance.

1 The beginning and ending points are transformed by the application and lie on the off-axis ellipse.

All other values are reserved and must not be used, as they may produce unpredictable results.

If the beginning and ending points are identical, regardless of whether or not they are on the ellipse, a full ellipse is drawn.

Return Value

GS_SUCC Successful.
GS_CORD Invalid coordinate.
GS_ELMM Invalid major or minor axis.
GS_INAC Virtual terminal inactive.
GS_ANGL Invalid angle.
GS_NMEM Insufficient resources.
GS_AEND Invalid end point.
GS_ASTR Invalid start point.

2.6.16 gsecnv

Purpose
Converts an ellipse to a polyline.

C Syntax

```
int gsecnv_ (cx, cy, ma, mi, ang, bx, by, ex, ey, rot, len, x, y, pre)

int *cx, *cy, *ma, *mi, *ang, *bx, *by, *ex, *ey, *rot, *len, *x, *y, *pre;
```

FORTRAN Syntax

```
INTEGER function gsecnv (cx, cy, ma, mi, ang, bx, by, ex, ey, rot, len, x, y,

INTEGER cx, cy, ma, mi, ang, bx, by, ex, ey, rot, len, x(*), y(*), pre
```

Pascal Syntax

```
FUNCTION gsecnv_ (

VAR cx, cy, ma, mi, ang, bx, by, ex, ey, rot, len: INTEGER;
VAR x, y: ARRAY [1..k] of INTEGER;
VAR pre: INTEGER
): INTEGER [PUBLIC];
```

Description
The **gsecnv** subroutine converts a counterclockwise elliptical arc definition into an array of vertices. The list of vertices can then be used to draw an elliptical arc with the **gspoly** subroutine or to fill an elliptical arc with the **gsfply** subroutine. In general, it can be concatenated with other list(s) of vertices to draw or fill more complex shapes, such as chord arcs, pie arcs, or rectangles with round corners.

When the beginning and ending points are identical, the list of vertices contains the full ellipse, which can then be drawn or filled.

Subtopics
2.6.16.1 Parameters

2.6.16.1 Parameters

cx, cy Define the coordinates of the center of the ellipse.

ma, mi Define half of the nonzero major and minor axes of the ellipse.

ang Defines the off-axis angle of the ellipse. If **ang** is 0, the major axis is the x-axis and the minor axis is the y-axis. A positive value rotates the ellipse counterclockwise; a negative value rotates it clockwise. All values are in degrees and modulo 360.

bx, by Define the coordinates of the beginning point of the arc.

ex, ey Define the coordinates of the ending point of the arc.

rot Specifies whether the application must perform rotational transformation. Possible settings are:

0 The coordinates of the beginning and ending points passed by the application correspond to an arc of an orthogonal ellipse. No rotational transformation is performed, thus improving performance.

1 The beginning and ending points are transformed by the application and lie on the off-axis ellipse.

All other values are reserved and must not be used, as they may produce unpredictable results.

len Defines the number of points in the coordinate **x** and **y** arrays. It must be numerically at least one greater than the value contained in the precision parameter and greater than or equal to 65.

x, y Define, as coordinate arrays, the vertices that represent the elliptical shape when drawn or filled.

pre Defines precision level, which specifies the maximum number of line segments that can be generated for a full ellipse. The number of line segments actually generated depends on the size of the ellipse.

There are four levels of precision that can be requested:

64 (65 vertices)

128 (129 vertices)

256 (257 vertices)

512 (513 vertices).

Therefore, **len** = **pre** + 1.

All other precision values are reserved and must not be used, as their results are unpredictable. The default value for **pre** is 64.

The subroutine allows ample leniency toward the accuracy of the

specification of the beginning and ending points. The arc of the specified angle always starts and ends exactly at the specified points. If the beginning and ending points are identical, a full ellipse of the specified angle is generated.

When the subroutine is invoked, the length parameter must contain the maximum number of entries in the **x** and **y** arrays. If erroneous conditions arise, **len** is set to 0. Under normal conditions, **len** specifies the number of vertices returned by the subroutine in the **x** and **y** arrays.

For Pascal, the application must declare the arrays passed as being fixed length and declare the routine as accepting arrays of that length; the **k** in the routine declaration must be a constant.

Return Value

GS_SUCC Successful.
GS_CORD Invalid coordinate.
GS_NCOR Invalid number of coordinates.

2.6.17 gsecur

Purpose
Erases the cursor, making it invisible.

C Syntax

int gsecur_ ( )

FORTRAN Syntax

INTEGER function gsecur

Pascal Syntax

FUNCTION gsecur_: INTEGER [**PUBLIC];**

Description
The **gsecur** subroutine makes the cursor invisible.

For adapters with hardware cursor support, **gsecur** simply turns off the cursor. Otherwise, **gsecur** reverses the actions taken to place the cursor in the frame buffer.

Return Value

GS_SUCC Successful.
GS_INAC Virtual terminal inactive.

2.6.18 gsell

Purpose
Draws an ellipse.

C Syntax

```
int gsell_ (cx, cy, ma, mi, ang)

int *cx, *cy, *ma, *mi, *ang;
```

FORTRAN Syntax

```
INTEGER function gsell (cx, cy, ma, mi, ang)

INTEGER cx, cy, ma, mi, ang
```

Pascal Syntax

```
FUNCTION gsell_ (

VAR cx, cy, ma, mi, ang : INTEGER
): INTEGER [PUBLIC];
```

Description

The **gsell** subroutine draws an ellipse of the specified axes and angle. The axes are expressed in number of pixels.

The angle specifications are given in tenths of degrees, from 0 to 3600. Values outside this range cause the **gsell** subroutine to fail.

The relevant attributes are:

- Color ma
- Plane mas
- Line color inde
- Line styl
- Logical operation

Subtopics
2.6.18.1 Parameters

2.6.18.1 Parameters

cx, cy Define the coordinates of the center of the ellipse.

ma, mi Define half of the nonzero major and minor axes of the ellipse.

ang Defines the angle between the major axis and the x-axis. If it is 0, the major axis is on the x-axis and the minor axis is on the y-axis. The angle is expressed in tenths of degrees, from 0 to 3600.

Return Value

GS_SUCC Successful.
GS_CORD Invalid coordinate.
GS_ELMM Invalid major or minor axis.
GS_INAC Virtual terminal inactive.
GS_ANGL Invalid angle.
GS_NMEM Insufficient resources.

2.6.19 gseply

Purpose
Defines the end of an area to fill.

C Syntax

int gseply_ ( )

FORTRAN Syntax

INTEGER function gseply

Pascal Syntax

FUNCTION gseply_ : INTEGER [**PUBLIC];**

Description

The **gseply** subroutine defines the end of a two-dimensional shape or set of shapes to be filled, then fills each of the valid primitives drawn since the last **gspcls** or **gsbply** subroutine was called.

The relevant attributes are:

Color ma
Plane mas
Fill color inde
Fill styl
Logical operation

Return Value

GS_SUCC Successful.
GS_USUC Unsuccessful.

Related Information
In this book: "gsbply" in topic 2.6.5 and "gspcls" in topic 2.6.41.

2.6.20 gsevds

Purpose
Disables the reporting of events.

C Syntax

int gsevds_ (event)

int *event;

FORTRAN Syntax

INTEGER function gsevds (event)

INTEGER event

Pascal Syntax

FUNCTION gsevds_ (

VAR event: INTEGER
): INTEGER [**PUBLIC];**

Description
The **gsevds** subroutine disables the reporting of events of a given type. When the keyboard event is disabled, the keyboard is locked and no keystroke input is placed in the input ring buffer. Similarly, for all other devices, if an event is disabled, the device producing the event is inhibited from placing input into the ring.

A valid input ring must be defined during the GSL initialization.

Subtopics
2.6.20.1 Parameters

2.6.20.1 Parameters

event The recognized events on the PS/2 are as follows:

1 Keystroke
3 Locator movement or button

The user can enable the keyboard by sending the **ESC b** key sequence from within a program to the output device.

Note: When the keyboard is disabled, this sequence cannot be entered.

Return Value

GS_SUCC Successful.
GS_EVNT Invalid event type.
GS_UNSC Unsuccessful.

2.6.21 gseven

Purpose
Enables the reporting of events.

C Syntax

```
int gseven_ (event)

int *event;
```

FORTRAN Syntax

```
INTEGER function gseven (event)

INTEGER event
```

Pascal Syntax

```
FUNCTION gseven_ (

VAR event: INTEGER
): INTEGER [PUBLIC];
```

Description
The **gseven** subroutine enables the reporting of events of a given type. If the device producing the event is enabled, then **gseven** lets it put data into the ring buffer. If the event type is not recognized, no action is taken.

A valid input ring must be defined during the GSL initialization.

Subtopics
2.6.21.1 Parameters

2.6.21.1 Parameters

event The recognized events on the PS/2 are as follows:

1 Keystroke
3 Locator movement or button

After GSL initialization, only the keyboard is enabled. If the application wishes the other input devices enabled, it must explicitly enable them with this command.

Return Value

GS_SUCC Successful.
GS_EVNT Invalid event type.
GS_UNSC Unsuccessful.

2.6.22 gsevwt

Purpose
Waits for an input event.

C Syntax

int gsevwt_ (wait, data)

int *wait, data[13];

FORTRAN Syntax

INTEGER function gsevwt (wait, data)

INTEGER wait, data (13)

Pascal Syntax

FUNCTION gsevwt_ (

VAR wait: INTEGER;
VAR data: ARRAY [**0..12**] **of INTEGER**
): INTEGER [**PUBLIC];**

Description
The **gsevwt** subroutine returns the relevant information for the oldest input event in the ring buffer.

The function works as follows:

If an event is in the ring, then **gsevwt** parses the oldest event in the ring. It returns the event type and its data in the buffer provided by the application.

If no event is in the ring and the application requested no wait **gsevwt** returns immediately. If the application requested a wait, the process execution is suspended until an enabled input event occurs; then **gsevwt** returns the event type and its data in the buffer provided.

Warning: **gsevwt** uses the application buffer passed to it for temporary storage. If the user has explicitly keyed part of an ANSI control sequence when the application calls **gsevwt** with no wait request, then **gsevwt** finds a partial event in the ring and leaves part of the parsed data for the event in the application buffer; however, **gsevwt** returns a timeout event class. Unless the application returns the same unmodified buffer, or a different buffer containing identical information, the results of the next call to **gsevwt** will be incorrect.

A valid input ring must be defined during the GSL initialization.

Subtopics
2.6.22.1 Parameters

2.6.22.1 Parameters

wait Determines whether or not to wait for an event. If **wait** is 0, then **gsevwt** does not wait for an event if no event is available.

data Specifies the location where GSL is to store the input data (up to 13 words). The **data** must be word aligned:

The possible events are:

1 Keystroke(s)

This event type occurs when the user types a single graphic character or a single-byte control character. For these two events, **gsevwt** returns a null-terminated byte string representing the graphic or control character that was typed. This event may also occur if the user has explicitly keyed an ANSI escape sequence; **gsevwt** returns two bytes, the ESC and the next character in the sequence.

The **data** consists of a null-terminated ASCII string and is structured as follows:

2 Control sequence

This event type indicates an ANSI control sequence, which is of the form:

ESC [**p ; p ;...p f]**

where ESC is the ASCII escape character, **p** represents a parameter (one or more ASCII digits), the ellipsis represents additional parameters separated by semicolons, and **f** represents the final character that terminates the sequence (ASCII **a-z** or **A-Z**).

The ANSI control sequence occurs when the user presses a program function key on the keyboard or if the user enters an explicit control sequence.

The data consists of the parsed control sequence information. The Final Character is the valid or invalid final character. The Count indicates the number of parameters in the control sequence, with a maximum count of 10. These fields are followed by the Parameters. The **data** is structured as follows:

3 Locator

This event indicates the user has moved the locator or pressed a button on the locator.

The data consists of locator position and status information. The X value and the Y value field contain a relative movement (delta x, delta y) for a mouse and an absolute position (x, y) for a tablet. The Timestamp, which is elapsed time since system startup (IPL), is in sixtieths of a second.

The Buttons field contains the locator button status. For a mouse, each bit corresponds to a button, the most significant bit representing Button 1. A bit set to 1 indicates that the

corresponding button is pressed. For a tablet, the most significant five bits represent the button pressed, according to the following scheme:

Status	**Button**
0	None pressed
1	Cursor upper left, stylus tip
2	Cursor upper right
3	Cursor lower left
4	Cursor lower right

For a tablet, the sixth most significant bit of the Buttons field indicates that the sensor is on (bit set) or off (bit not set).

The Type field contains a 0 if the locator is a mouse and a 1 if the locator is a tablet. The **data** is structured as follows:

4 LPFK

This event type occurs when the user presses a key on the LPFK.

The data consists of the LPFK information. The LPFK field contains the decimal number 0 through 31 of the LPFK pressed by the user. The Timestamp (time since system startup) is in sixtieths of a second. The **data** is structured as follows:

5 Valuator

This event type occurs when the user turns a valuator dial.

The data consists of the valuator information. The Valuator field contains the decimal number 0 through 7 of the valuator turned by the user. The Valuator Delta field contains the difference between the current valuator value and the last valuator value. The delta for a full turn is 256 for the IBM Valuator. The delta is positive for clockwise rotation and negative for counterclockwise rotation. The Timestamp (time since system startup) is in sixtieths of a second. The **data** is structured as follows:

6 Key Code

This event type occurs when the virtual terminal is in non-translated mode **and** a keyboard key is pressed, held down, or released. The **data** is structured as follows:

Key position codes are found under "keyboard" in topic 2.5.13.

7 Pick Event

This event type occurs while the pick operation is enabled and graphics primitives are being sent to the adapter. The **data** is structured as follows:

A pick event code is generated when a structure traversal occurs. The pick occurs when pixels intersect the pick window

(defined by the pick enable window size). The detection mode is always immediate, so that an event is generated as soon as an event occurs.

The pick event type is provided only for use with the IBM 5081 Display Adapter, and not for use with other displays.

10 Timeout

No data is returned.

It is important to note that **gsevwt** does not detect ANSI escape sequences. However, with the default virtual terminal keyboard mapping, it is not possible to generate an escape sequence by pressing a single key. Because **gsevwt** does parse ANSI control sequences, the routine cannot consider the press of the escape key an event, so the routine waits for the next character to decide if the escape implies the start of a control sequence. Only if the next character is not the left bracket does **gsevwt** return the escape and the next character.

If the return code indicates overflow, the most recent input events from enabled devices are lost.

Return Value

GS_SUCC Successful.
GS_ROVR Ring buffer overflow.
GS_UDRG Ring undefined.
GS_PARM Invalid number of escape parameters.
GS_ICTL Invalid final character.

Related Information
In this book: "keyboard" in topic 2.5.13.

2.6.23 gsfatt

Purpose
Sets the fill attributes.

C Syntax

```
int gsfatt_ (color, pattern, reserved)

int *color, *pattern, *reserved;
```

FORTRAN Syntax

```
INTEGER function gsfatt (color, pattern, reserved)

INTEGER color, pattern, reserved
```

Pascal Syntax

```
FUNCTION gsfatt_ (

VAR color, pattern, reserved: INTEGER
): INTEGER [PUBLIC];
```

Description
The **gsfatt** subroutine defines the attributes for the class of fill functions, which includes **gsfci**, **gsfell**, **gsfrec**, and **gsfply**.

Subtopics
2.6.23.1 Parameters

2.6.23.1 Parameters

color Refers to an entry in the color map. If **color** is -1, the attribute is unchanged. The default color after initialization is 7, white.

pattern Contains a value from the following list:

Value	Display	Printer or Plotter
-1	No change	No change
0	Solid	Solid
1	Horizontal lines	Narrow right diagonal lines
2	Vertical lines	Medium right diagonal lines
3	135-degree lines	Wide right diagonal lines
4	45-degree lines	Narrow diagonal cross-hatched
5	Cross-hatched (horizontal and vertical lines)	Medium diagonal cross-hatched
6	Cross-hatched (45- and 135-degree lines)	Wide diagonal cross-hatched

The default pattern is solid (0).

Some printers and plotters support additional fill patterns that can be selected with a **pattern** index greater than 6. If the device you are using does not support additional fill patterns and you specify a **pattern** index greater than 6, then the **gsfatt** subroutine returns the value **GS_SYLI**.

reserved Represents a parameter that **gsfatt** ignores.

Return Value

GS_SUCC Successful.
GS_COLI Invalid color index.
GS_SYLI Invalid style index.

2.6.24 gsfci

Purpose
Fills a circle.

C Syntax

int gsfci_ (cx, cy, cr)

int *cx, *cy, *cr;

FORTRAN Syntax

INTEGER function gsfci (cx, cy, cr)

INTEGER cx, cy, cr

Pascal Syntax

FUNCTION gsfci_ (

VAR cx, cy, cr : INTEGER
): INTEGER [**PUBLIC];**

Description

The **gsfci** subroutine fills a circle of the specified radius. The radius is expressed in number of pixels.

The relevant attributes are:

    Color ma
    Plane mas
    Fill color inde
    Fill pattern inde
    Logical operation

Subtopics
2.6.24.1 Parameters

2.6.24.1 Parameters

cx, cy Define the coordinates of the center of the circle.

cr Defines the radius of the circle.

If the radius is 0, a single point is filled at the center.

If the radius is 0, a single point is filled at the center.

Return Value

GS_SUCC Successful.
GS_CORD Invalid coordinate.
GS_RDUS Invalid radius specification.
GS_INAC Virtual terminal inactive.

2.6.25 gsfell

Purpose
Fills an ellipse.

C Syntax

int gsfell_ (cx, cy, ma, mi, ang)

int *cx, *cy, *ma, *mi, *ang;

FORTRAN Syntax

INTEGER function gsfell (cx, cy, ma, mi, ang)

INTEGER cx, cy, ma, mi, ang

Pascal Syntax

FUNCTION gsfell_ (

VAR cx, cy, ma, mi, ang : INTEGER
): INTEGER [**PUBLIC];**

Description

The **gsfell** subroutine fills an ellipse of the specified axes and angle. The axes are expressed in number of pixels.

The angle specifications are given in tenths of degrees, from 0 to 3600. Values outside this range cause the **gsfell** subroutine to fail.

The relevant attributes are:

    Color ma
    Plane mas
    Fill color inde
    Fill pattern inde
    Logical operation

Subtopics
2.6.25.1 Parameters

2.6.25.1 Parameters

cx, cy Define the coordinates of the center of the ellipse.

ma, mi Define half of the nonzero major and minor axes of the ellipse.

ang Defines the angle between the major axis and the x-axis. If it is 0, the major axis is on the x-axis and the minor axis is on the y-axis. The angle is defined in tenths of degrees, from 0 to 3600, specified in a counterclockwise direction.

Return Value

GS_SUCC Successful.
GS_CORD Invalid coordinate.
GS_ELMM Invalid major or minor axis.
GS_INAC Virtual terminal inactive.
GS_ANGL Invalid angle.
GS_NMEM Insufficient resources.

2.6.26 gsfply

Purpose
Draws a filled polygon.

C Syntax

```
int gsfply_ (number, x, y)

int *number, *x, *y;
```

FORTRAN Syntax

```
INTEGER function gsfply (number, x, y)

INTEGER number
INTEGER x (*)
INTEGER y (*)
```

Pascal Syntax

```
FUNCTION gsfply_ (

VAR number: INTEGER;
VAR x, y: ARRAY [1..k] of INTEGER
): INTEGER [PUBLIC];
```

Description
The **gsfply** subroutine fills an area that is described by the points defined in the **number** and **x, y** parameters, with the color determined by the last call to the **gsfatt** subroutine.

The relevant attributes are:

- Color ma
- Plane mas
- Fill color inde
- Fill pattern inde
- Logical operation

Subtopics
2.6.26.1 Parameters

2.6.26.1 Parameters

number Defines the number of points in the coordinate arrays. This value must be 3 or more.

x, y Define, as coordinate arrays, the points surrounding the polygon to fill.

The edges are treated as part of the area to be filled.

The **gsfply** subroutine fills a closed polygon with a pattern, generated by creating an edge between the first and the last points. The first and the last points described by the parameters may be equal, but it is not required and is actually less efficient.

For Pascal, the application must declare the arrays passed as being fixed length and declare the routine as accepting arrays of that length; that is, the **k** in the routine declaration must be a constant.

Return Value

GS_SUCC Successful.
GS_CORD Invalid coordinate.
GS_NCOR Invalid number of coordinates.
GS_NMEM Insufficient resources.
GS_INAC Virtual terminal inactive.

2.6.27 gsfrec

Purpose
Draws a filled rectangle.

C Syntax

```
int gsfrec_ (x1, y1, x2, y2)

int *x1, *y1, *x2, *y2;
```

FORTRAN Syntax

```
INTEGER function gsfrec (x1, y1, x2, y2)

INTEGER x1, y1, x2, y2
```

Pascal Syntax

```
FUNCTION gsfrec_ (

VAR x1, y1, x2, y2: INTEGER
): INTEGER [PUBLIC];
```

Description
The **gsfrec** subroutine fills the rectangular area defined by the lower leftmost and upper rightmost coordinate parameters, with the color determined by the last call to the **gsfatt** subroutine.

The relevant attributes are:

- Color ma
- Plane mas
- Fill color inde
- Fill pattern inde
- Logical operation

Subtopics
2.6.27.1 Parameters

2.6.27.1 Parameters

x1, y1 Define the lower left corner of the rectangular area to fill.

x2, y2 Define the upper right corner of the rectangular area to fill.

The edges of the rectangle are treated as part of the area to be filled.

Return Value

GS_SUCC Successful.
GS_CORD Invalid coordinate.
GS_INAC Virtual terminal inactive.

2.6.28 gsgtat

Purpose
Sets the attributes for the geometric text drawing functions.

C Syntax

```
int gsgtat_ (color, baseline, pre, expan, spac, height,
             upvectx, upvecty, alignhz, alignvt, font_ID, font)

int *color, *baseline, *pre, *expan, *spac, *height,
int *upvectx, *upvecty, *alignhz, *alignvt, *font_ID;
char *font;
```

FORTRAN Syntax

```
INTEGER function gsgtat (color, baseline, pre, expan, spac, height,
upvectx, upvecty, alignhz, alignvt, font_ID, font)

INTEGER color, baseline, pre, expan, spac, height
INTEGER upvectx, upvecty, alignhz, alignvt, font_ID
CHARACTER*n font
```

Pascal Syntax

```
FUNCTION gsgtat_ (

VAR color, baseline, pre, expan, spac, height: INTEGER;
VAR upvectx, upvecty, alignhz, alignvt, font_ID: INTEGER;
VAR font: ARRAY [0..k] of CHAR
): INTEGER [PUBLIC];
```

Description
The **gsgtat** subroutine defines the attributes and fonts for the geometric text drawing functions.

Note: The attributes defined by this command are applicable only to geometric text.

Subtopics
2.6.28.1 Parameters

2.6.28.1 Parameters

color Specifies an entry in the color map for text color. If it is -1, the attribute is unchanged.

baseline Determines the direction of the geometric text drawing. The valid values are:

-1 Attribute remains unchanged.

0 Specifies 0 degrees, or left to right in the viewer's terms.

1 Specifies 90 degrees, or up in the viewer's terms.

2 Specifies 180 degrees, or right to left in the viewer's terms.

Note: The characters appear upside down.

3 Specifies 270 degrees, or down in the viewer's terms.

Note: The **baseline** parameter does not change character rotation. Use the **upvectx** and **upvecty** parameters to rotate text.

pre Specifies the desired text precision used in drawing text primitives. The valid values are:

-1 Attribute remains unchanged.

1 Character precision.

2 Stroke precision.

expan Defines as a 32-bit fractional integer the deviation of the width/height ratio of the character from the ratio defined in the font. The expansion factor only changes the width of the character.

In the above figure, the first 16 bits contain zeros, S represents the sign bit, INTEGER represents the integer portion of the width/height ratio, and FRACTION represents the fractional portion of the ratio. A 32-bit integer value of 0x80000000 indicates that this attribute is unchanged.

spac Specifies the character spacing, or additional number of pixels to be inserted between characters. The value is a 16-bit signed integer. The preferred value for this parameter varies, based on the display in use. The maximum value that is allowed is equal to the display width in pixels. A value of 0x80000000 for this parameter indicates that the attribute is unchanged.

height Specifies the current character height for geometric text in pixels. This value is defined as a 16-bit signed integer, with the maximum value equal to the height of the display in pixels. A value of 0x80000000

for this parameter indicates that the attribute is unchanged.

upvectx, upvecty Specify the x and y coordinates for the up direction of a character or text string. The valid range for these values is ± the display dimensions in pixels. A value of 0x80000000 for this parameter indicates that the attribute is unchanged.

The up vector is a two-dimensional vector on the text plane, specified by the current text draw. (The origin of the vector is defined by the geometric text command, **gsgtxt**.) Only the direction, not the length, of the vector is relevant.

alignhz Specifies the horizontal alignment of the text for subsequent text drawing. Values are as follows:

```
-1   Attribute is unchanged
 1   Normal
 2   Left
 3   Center
 4   Right
```

alignvt Specifies the vertical alignment of the text for subsequent text drawing. Values are as follows:

```
-1   Attribute is unchanged
 1   Normal
 2   Top
 3   Cap
 4   Half
 5   Base
 6   Bottom
```

font_ID Specifies the ID of the font as a 32-bit integer, which defines the type of font to use. This ID is determined by the user while defining each geometric font. Possible values are:

-1	A **font_ID** has been defined in a previous call to the **gsgtat** subroutine, and this attribute is unchanged.
1025 to 32767	These values are used to specify 1-byte geometric fonts, and refer to a value defined in each geometric font file.
32768 to 65535	These values are used to specify 2-byte geometric fonts, and refer to a value defined in each geometric font file.

Only 1 **font_ID** is active at any time. To change the **font_ID**, **gsgtat** must be called again with new **font_ID** and **font** parameters. When a new **font_ID** is specified, the previous **font_ID** is purged from the font table.

For 2-byte geometric text, up to 128 segment IDs can be used per **font_ID**.

When used with the **font** parameter, the **font_ID** is associated with the **font** used for font selection.

font Contains the null-terminated full path name of the file used when the font attribute is specified as user. If a **font_ID** is defined, this parameter must also be defined. A value of -1 for this parameter indicates that the attribute is unchanged. For information on the format of font files for geometric text, see "Geometric Text Font Format" in topic 2.3.19.2.

Attributes are only valid for the currently active font.

This subroutine must be called before the **gsgtxt** subroutine or an error results.

For Pascal, the application must declare the arrays passed as being fixed length and declare the routine as accepting arrays of that length. The **k** in the routine declaration must be a constant.

Return Value

GS_SUCC Successful.
GS_COLI Invalid color index.
GS_PREC Invalid text precision value.
GS_EXPN Invalid character expansion factor.
GS_FNTN Invalid file name.
GS_INSV Invalid spacing value.
GS_BASL Invalid baseline direction.
GS_HIGH Invalid height value.
GS_UPVT Invalid up vector value.
GS_ALGN Invalid alignment value.

Related Information
In this book: "fonts" in topic 2.3.19, "Geometric Text Font Format" in topic 2.3.19.2, and "gsgtxt" in topic 2.6.29.

2.6.29 gsgtxt

Purpose
Writes geometric text.

C Syntax

```
int gsgtxt_ (x, y, number, text)

int *x, *y, *number;
char *text;
```

FORTRAN Syntax

```
INTEGER function gsgtxt (x, y, number, text)

INTEGER x, y, number
CHARACTER*n text
```

Pascal Syntax

```
FUNCTION gsgtxt_ (

VAR x, y, number: INTEGER;
VAR text: ARRAY [1..k] of CHAR
): INTEGER [PUBLIC];
```

Description
The **gsgtxt** subroutine writes geometric characters starting at the baseline position defined by the parameters and writes the number of characters indicated by the parameters according to the relevant attributes.

The relevant attributes are:

- Color ma
- Plane mas
- Fon
- Text color inde
- Character expansion facto
- Character spacin
- Character heigh
- Character up vecto
- Character alignmen
- Baseline direction

Subtopics
2.6.29.1 Parameters

2.6.29.1 Parameters

x, y	Define the coordinates of the baseline position for writing geometric text.
number	Indicates the number of bytes to write from the **text** string. The maximum number of characters allowed is 1024 for single byte fonts and 512 for two-byte fonts, which is determined by the display and font in use.
text	Contains the N-bit ASCII codes for the characters to write, as an array.

Return Value

GS_SUCC	Successful.
GS_CORD	Invalid coordinate.
GS_FBUF	Frame buffer overflow.
GS_INAC	Virtual terminal inactive.
GS_NOFT	Font not loaded.
GS_NOAT	Text attribute not set.

Related Information
In this book: "fonts" in topic 2.3.19, "Geometric Text Font Format" in topic 2.3.19.2, "gsgtat" in topic 2.6.28, and "gsqgtx" in topic 2.6.47.

2.6.30 gsinit

Purpose
Initializes the GSL subroutines.

C Syntax

```
int gsinit_ (buffer, size, save_restore, f_grant, f_retract, fildes)

int *buffer, *size, *save_restore;
int (*f_grant) ( ), (*f_retract) ( );
int *fildes;
```

FORTRAN Syntax

```
INTEGER function gsinit (buffer, size, save_restore, f_grant, f_retract, filde

INTEGER buffer (*), size, save_restore, fildes
EXTERNAL f_grant, f_retract
```

Pascal Syntax

```
FUNCTION gsinit_ (

VAR buffer: ARRAY [0..k] of INTEGER;
VAR size, save_restore, f_grant, f_retract, fildes: INTEGER
): INTEGER [PUBLIC];
```

Description
The **gsinit** subroutine initializes the GSL. It allocates any private storage required, and sets attributes to the default values where necessary. It also forces the virtual terminal of the application to Monitor Mode and sets up the signal processing routines for the **SIGRETRACT** and **SIGGRANT** signals, and optionally, the **SIGMSG** signal.

Subtopics
2.6.30.1 Parameters

2.6.30.1 Parameters

buffer Defines the Monitor Mode input ring buffer to be used by the GSL input functions. **buffer** must be word aligned and at least 128 bytes long. For output to a printer or plotter device, set the **buffer** parameter to -1. (In C, **buffer** is a pointer to an integer containing the value -1. In Pascal, it is a variable containing the value -1.)

size Defines the length of **buffer** in bytes. Depending on the value of **size**, **gsinit** performs the following actions:

size=0 The GSL ignores the **buffer** parameter and does not provide input support. The application must provide a means for receiving input events and can use the **read** system call or set up its own ring buffer mechanism.

size<128 The **gsinit** subroutine does not initialize the GSL.

size=128 The GSL establishes the virtual terminal linkage to the input ring buffer provided by the application and provides input support and sets up a **SIGMSG** signal catcher.

save_restore Determines whether to save the display frame buffer and adapter states.

If **save_restore** is nonzero, the GSL saves the current contents of the display frame buffer as well as the current adapter state when the virtual terminal must become inactive and restores both the frame buffer contents and adapter state when it becomes active.

If **save_restore** is 0, the GSL saves only the adapter state and assumes that the application either saves the frame buffer or reconstructs it in some fashion.

f_grant Sets up processing of the **SIGGRANT** signal. If **f_grant** is nonzero, it is assumed to be the address of an application supplied function, and the GSL calls the function as part of the **SIGGRANT** signal handling. If **save_restore** is nonzero, the application function is called before the frame buffer is restored.

f_retract Sets up processing of the **SIGRETRACT** signal. If **f_retract** is nonzero, it is assumed to be the address of an application supplied function, and the GSL calls the function as part of the **SIGRETRACT** signal handling.

fildes Determines where output is directed. The output device is specified by one of the following:

The value -1, which specifies standard output.

A file descriptor returned by a **creat**, **open**, **dup**, or **fcntl** system call.

A null-terminated character string up to 11 characters long, which names an environment variable defining a printer or plotter device. In this case, the value of the **buffer** parameter must be -1. (See "Printers and Plotters" in topic 2.6.4.2.)

(In C, **fildes** is a pointer to a file descriptor, an integer, or a character string. In Pascal, it is a variable containing one of these values.)

If the initialization process is unsuccessful, the virtual terminal is not placed in Monitor Mode and invocation of any other GSL routines will cause unpredictable results.

For printers or plotters, if initialization is unsuccessful, the application can either terminate or re-drive the initialize function with a valid character string as a means of correcting the problem.

For Pascal, the application must declare the arrays passed as being fixed length and declare the routine as accepting arrays of that length; that is, the **k** in the routine declaration must be a constant.

Pascal cannot directly provide the address of a routine. An assembler function may be used to derive the address of a routine passed to the GSL.

The **f_grant** and **f_retract** routines supplied by the application are called on the signal level and **must return**. These application routines must not use either **setjmp** or **longjmp** subroutines.

The GSL supports use of the **sdb** symbolic debugger on the RT PC and the **dbx** symbolic debugger on the PS/2 by redirection to a supplied file descriptor. If two virtual terminals are open and the GSL application runs on one, the application may get the file descriptor for the second and supply that descriptor at GSL initialization. The GSL directs its output to the second virtual terminal while **sdb** on the RT PC and **dbx** on the PS/2 directs its output to the first; either is activated in the standard manner.

The user routine called at **SIGGRANT** can be called before **gsinit** returns to the application.

Return Value

GS_SUCC Successful.
GS_HBUS Cannot access hardware bus.
GS_ADPT Invalid display type.
GS_FONT Cannot access default font.
GS_RING Buffer too small.
GS_HDCP Invalid file descriptor for hard copy output.
GS_HDLK Unable to create lock file.
GS_HDIM Insufficient memory.
GS_HDDB Device is busy.
GS_HDNA Physical device not attached.
GS_HDMG Maximum number of graphics devices open.
GS_HDIF No system interprocess communication buffers left.
GS_HDSF The **fork** system call failed.
GS_HDGO Specified graphics device already open.
GS_HDGN Specified graphics device does not exist.
GS_HDGU Specified graphics device driver is unknown.

Related Information
In this book: "Printers and Plotters" in topic 2.6.4.2.

2.6.31 gslatt

Purpose
Sets the line attributes.

C Syntax

```
int gslatt_ (color, style)

int *color, *style;
```

FORTRAN Syntax

```
INTEGER function gslatt (color, style)

INTEGER color, style
```

Pascal Syntax

```
FUNCTION gslatt_ (

VAR color, style: INTEGER
): INTEGER [PUBLIC];
```

Description
The **gslatt** subroutine defines the attributes for the class of line drawing functions.

Subtopics
2.6.31.1 Parameters

2.6.31.1 Parameters

color Refers to a line color entry in the color map. If it is -1, the attribute is unchanged. The default color is 7, white.

style Sets or resets the line attributes. The line value of **style** can be one of the following:

Value	Display	Printer or Plotter
-1	No change	No change
0	Solid	Solid
1	Dash	Dash
2	Dot	Dot
3	Dash-dot	Dash-dot
4	Dash-dot-dot	Dash-dot-dot
50	User-supplied	Not available
100	Continuous solid	Solid
101	Continuous dash	Dash
102	Continuous dot	Dot
103	Continuous dash-dot	Dash-dot
104	Continuous dash-dot-dot	Dash-dot-dot
150	Continuous user-supplied	Not available

The default attribute is solid (0).

The GSL supplied line style patterns are implemented in a device-dependent fashion. All line style indices not described above are reserved.

For line styles 1-4 and line style 50 the GSL line drawing functions ensure that a line or line segment starts and ends with a run of the line color. For these line styles, the GSL does not continue the pattern from one polyline segment to another.

For line styles 101-104 and line style 150, the GSL continues the pattern across multiple lines or line segments until the application makes another call to **gslatt** to reset the line pattern. In this case, unlike styles 1-4 and 50, the GSL continues the pattern from one polyline segment to another. Continuous line styles are not available on printers and plotters.

Return Value

GS_SUCC Successful.
GS_COLI Invalid color index.
GS_SYLI Invalid style index.

2.6.32 gslcat

Purpose
Sets the locator attributes.

C Syntax

```
int gslcat_ (hg, vg)

int *hg, *vg;
```

FORTRAN Syntax

```
INTEGER function gslcat (hg, vg)

INTEGER hg, vg
```

Pascal Syntax

```
FUNCTION gslcat_ (

VAR hg, vg: INTEGER
): INTEGER [PUBLIC];
```

Description
The **gslcat** subroutine sets the locator attributes. Its effect depends on the type of locator attached. For a mouse, **gslcat** sets the thresholds. For a tablet, it sets the dead zone.

Subtopics
2.6.32.1 Parameters

2.6.32.1 Parameters

hg, vg Define the horizontal and vertical values for the locator threshold or dead zone, in units of 0.25 millimeter.

The mouse ***thresholds*** determine the granularity of input events reported, or the amount of horizontal or vertical mouse movement required before an event occurs.

The tablet ***dead zone*** is an area of the tablet in which no event reports occur, even if the tablet sensor is present. This dead zone allows the application to make the tablet aspect ratio compatible with the display and allows tablets of different sizes to appear the same size to an application. The dead zone acts as a border around the tablet. The device driver reports movement only when the x value is greater than or equal to **hg** or less than or equal to (maximum tablet value - **hg**), and the y value is greater than or equal to **vg** or less than or equal to (maximum tablet value - **vg**).

An attempt to set the locator attributes may fail for a variety of reasons, the most likely of which is that the device is not attached. The nature of the problem can be determined with a specific **ioctl** to the virtual terminal. (See "hft" in topic 2.5.11 for more information.)

Note that the **gslcat** subroutine allows an application to set the mouse thresholds or the tablet dead zone such that no events occur even if the device is enabled.

Return Value

GS_SUCC Successful.
GS_USUC Unsuccessful.

Related Information
In this book: "hft" in topic 2.5.11.

2.6.33 gsline

Purpose
Draws a line between two points.

C Syntax

```
int gsline_ (x1, y1, x2, y2)

int *x1, *y1, *x2, *y2;
```

FORTRAN Syntax

```
INTEGER function gsline (x1, y1, x2, y2)

INTEGER x1, y1, x2, y2
```

Pascal Syntax

```
FUNCTION gsline_ (

VAR x1, y1, x2, y2: INTEGER
): INTEGER [PUBLIC];
```

Description
The **gsline** subroutine draws a line, as defined by the current relevant attributes, from the first point to the second point defined by the parameters.

The relevant attributes are:

- Color ma
- Plane mas
- Line color inde
- Line styl
- Logical operation

Subtopics
2.6.33.1 Parameters

2.6.33.1 Parameters

x1, y1	Defines the coordinates of one end point of the line drawn by **gsline**.
x2, y2	Defines the coordinates of the second point of the line drawn by **gsline**.

Return Value

GS_SUCC	Successful.
GS_CORD	Invalid coordinate.
GS_INAC	Virtual terminal inactive.

2.6.34 gslock

Purpose
Postpones signal processing.

C Syntax

int gslock_ ( )

FORTRAN Syntax

INTEGER function gslock ( )

Pascal Syntax

FUNCTION gslock_ ( ): INTEGER [**PUBLIC];**

Description
The **gslock** subroutine causes the GSL not to acknowledge the **SIGRETRACT** signal, if it occurs, until the application requests resumption of the signal handling with the **gsunlk** subroutine. This permits the application to access the display frame buffer directly.

If the virtual terminal is inactive when the application calls **gslock** and the GSL has been instructed to save the frame buffer when the virtual terminal becomes inactive, **gslock** suspends the application until the virtual terminal becomes active and then returns a successful return code. If the GSL has been instructed not to save the frame buffer, **gslock** returns the **GS_INAC** return code immediately. The application must not access the display frame when **GS_INAC** is returned.

Note: If **SIGRETRACT** signal processing is suspended for more than 30 seconds, it is possible that a generated **SIGRETRACT** signal may be suspended long enough for the **SIGKILL** signal to occur, terminating the application process.

Return Value

GS_SUCC Virtual terminal active, safe to write to frame buffer.
GS_INAC Virtual terminal inactive.

2.6.35 gslop

Purpose
Specifies the logical operation used when drawing lines.

C Syntax

int gslop_ (operation)

int *operation;

FORTRAN Syntax

INTEGER function gslop (operation)

INTEGER operation

Pascal Syntax

FUNCTION gslop_ (

VAR operation INTEGER;
): INTEGER [**PUBLIC];**

Description
The **gslop** subroutine specifies the logical operation used for drawing the GSL line-oriented, fill, save/restore, and polymarker primitives. It does not apply to the text primitives.

Subtopics
2.6.35.1 Parameters

2.6.35.1 Parameters

operation Indicates the logical operation to perform between the primitive being drawn and the current contents of the frame buffer.

In the following table, please note:

The source pixels represent bits of data to be merged in some way with the corresponding bits of data in the destination rectangle.

The first three columns of the table specify the operations you can perform, and the **Code** column contains the corresponding value you should specify for the **operation** parameter.

A ~ (tilde) represents the logical INVERSE.

Type of Source	**Logical Operation**	**Type of Destination**	**Code**
		Destination clear	0
		Set Destination	15
	No operation	Destination	5
		~Destination	10
Source	REPLACE	Destination	3
Source	AND	Destination	1
Source	AND	~Destination	2
Source	Exclusive-or	Destination	6
Source	OR	Destination	7
Source	OR	~Destination	11
~Source	REPLACE	Destination	12
~Source	AND	Destination	4
~Source	AND	~Destination	8
~Source	Exclusive-or	Destination	9
~Source	OR	Destination	13
~Source	OR	~Destination	14

Replace (3) is the default logical operation.

Only REPLACE, AND, OR, and Exclusive-or (codes 3, 1, 7, and 6, respectively), are supported for VGA displays.

For printers, the operations performed are the same as those for displays, except that a value of 0 turns the color off, and a value of 15 changes the color to white.

For plotters, the default and only valid logical operation is logical OR. The plotters interpret this as an overstrike. The GSL performs each of the boolean operations for each bit of the source and destination color values enabled by the plane mask. The destination receives the color value that results from the operation.

The logical operations are performed on the color index rather than the color itself. This can cause some operations on color displays to produce results that are not expected.

Return Value

GS_SUCC Successful.
GS_LONS Logical operation not supported.

2.6.36 gsmask

Purpose
Defines planes to be modified.

C Syntax

```
int gsmask_ (mask)

int *mask;
```

FORTRAN Syntax

```
INTEGER function gsmask (mask)

INTEGER mask
```

Pascal Syntax

```
FUNCTION gsmask_ (

VAR mask: INTEGER
): INTEGER [PUBLIC];
```

Description
The **gsmask** subroutine defines the planes actually modified by the line, text, and fill functions.

Subtopics
2.6.36.1 Parameters

2.6.36.1 Parameters

mask	Indicates which planes of the display adapter frame buffer can be modified by the output functions. The most significant bits of the input are used to set the plane mask.

Return Value

GS_SUCC	Successful.
GS_INAC	Virtual terminal inactive.

2.6.37 gsmatt

Purpose
Sets the polymarker attribute.

C Syntax

```
int gsmatt_ (color, style, width, height, pattern, 0x, 0y)

int *color, *style, *width, *height, *pattern, *0x, *0y;
```

FORTRAN Syntax

```
INTEGER function gsmatt (color, style, width, height, pattern, 0x, 0y)

INTEGER color, style, width, height, pattern, 0x, 0y
```

Pascal Syntax

```
FUNCTION gsmatt_ (

VAR color, style, width, height: INTEGER;
pattern: ARRAY [1..k] of INTEGER;
0x, 0y: INTEGER
): INTEGER [PUBLIC];
```

Description
The **gsmatt** subroutine defines the marker for the GSL.

Subtopics
2.6.37.1 Parameters

2.6.37.1 Parameters

color Refers to a marker color entry in the color map. If it is -1, the attribute is unchanged. The default value for color is **7**, white.

style Defines the polymarker **style** as one of the following:

Value	Display	Printer or Plotter
-1	No change	No Change
0	User-defined (by **width**, **height**, **pattern**, **0x**, **0y**)	Not available
1	Dot (filled circle)	Point
2	Plus (+)	Plus (+)
3	Asterisk (*)	Asterisk (*)
4	Circular shape	Square shape
5	Cross (¦)	Cross (¦)
6	Unfilled box	Diamond

width, height Define in pixels the width and the height of the bit pattern to be used as the marker. If **width** or **height** equals -1, then the pattern remains unchanged.

pattern Defines the image used as a marker. The ceiling of (**width** / 32) indicates the number of words per row and **height** indicates the number of rows. The marker data must be supplied in row (scan line) major order. If **width** implies partial use of a word, the rest of the word is unused. To fully define the marker pattern, **pattern** should be (ceiling ¦ **height**) words in length.

0x, 0y Indicate the coordinates of the origin of the marker relative to the lower leftmost corner (0, 0) of the marker pattern. The origin must be placed inside the marker pattern, so that **0x** < **width** and **0y** < **height**. The origin of the marker is placed at the position indicated when the application places a marker with the **gsplym** subroutine. (See "gsplym" in topic 2.6.42.) If **0x** equals -1, then the origin remains unchanged.

The maximum size of the marker is device dependent. It equals the height and width of the display, which may be determined by calling the **gsqdsp** subroutine.

Note: The GSL subroutines do not make a copy of a user-defined polymarker. Changes or reuse of the storage where a user-defined shape is in use can cause unpredictable results.

For Pascal, the application must declare the arrays passed as being fixed length and declare the routine as accepting arrays of that length. The **k** in the routine declaration must be a constant.

Return Value

GS_SUCC Successful.
GS_COLI Invalid color index.
GS_PMSZ Marker size invalid.

GS_PMOR Marker origin invalid.
GS_PMSY Marker style invalid.

Related Information
In this book: "gsplym" in topic 2.6.42.

2.6.38 gsmcat

Purpose
Sets the cursor attributes.

C Syntax

int gsmcat_ (foreground, background, width, height, pattern, mask, 0x, 0y, log

int *foreground, *background, *width, *height, *pattern, *mask, *0x, *0y, *log

FORTRAN Syntax

INTEGER function gsmcat_ (foreground, background, width, height,
pattern, mask, 0x, 0y, logop)

INTEGER foreground, background, width, height, pattern, mask, 0x, 0y, logop)

Pascal Syntax

FUNCTION gsmcat_ (

VAR foreground, background, width, height: INTEGER;
VAR pattern: ARRAY [**1..k**] **of INTEGER;**
VAR mask: ARRAY [**1..k**] **of INTEGER;**
VAR 0x, 0y, logop: INTEGER
): INTEGER [**PUBLIC**];

Description
The **gsmcat** subroutine defines the multicolor cursor for the GSL. The **gscmap** subroutine must initialize the color map before **gsmcat** can be called.

Only one cursor, either the multicolor cursor or the single-color cursor, can be active in the GSL at any one time. The **gsmcat** subroutine forces all subsequent calls to the **gsmcur** and **gsecur** subroutines to operate on the multicolor version of the cursor. To change from the single-color cursor to the multicolor cursor, erase the cursor with **gsecur**, then call the **gsmcat** subroutine.

The multicolor cursor is a two-color, clipped cursor with logical operations. Its size is limited to 32 bits in width and 32 bits in height. Although the cursor origin cannot be moved outside the frame buffer boundaries, any portion beyond the origin that falls outside the frame buffer is clipped. In addition, a mask is provided that can be used to allow portions of the frame buffer to **show through** the cursor. Any bits set to 0 in the mask indicate that the matching bits in the cursor pattern do not affect the underlying frame buffer.

Subtopics
2.6.38.1 Parameters

2.6.38.1 Parameters

foreground Defines a color entry in the color map. This color is used for the foreground color (bits set to 1) in the multicolor cursor raster. A value of -1 indicates no change to this attribute.

background Defines a color entry in the color map. This color is used for the background color (bits set to 0) in the multicolor cursor raster. A value of -1 indicates no change to this attribute.

width, height Define, in pixels, the width and height of the bit pattern and mask to be used as the cursor. The maximum value for width and height of the cursor is 32 bits. If **width** or **height** equals -1, then the pattern and the mask remain unchanged.

pattern Defines the raster image used as a cursor. It must be specified in 32-bit integers, and there must be **height** number of rows. The GSL will only use **width** number of bits in each integer.

mask Defines the mask pattern of the cursor. Each bit in the **mask** corresponds with a bit in the multicolor cursor **pattern**. If a bit is set (has a value of 1), the matching bit in the pattern is applied to the underlying display raster. If a bit is not set (has a value of 0), the matching bit in the pattern is masked and does not affect the underlying display raster. The size of the **mask** must match the size of the **pattern** exactly.

0x, 0y Indicate the origin of the cursor relative to the lower leftmost corner (0, 0) of the cursor pattern. The origin must be placed within the cursor pattern: **0x** < **width** and **0y** < **height**. The origin of the cursor is placed at the position indicated, when the application moves the cursor using the **gsmcur** subroutine. If **x** equals -1, then the origin remains unchanged.

logop Defines the logical operation to perform between the cursor pattern being drawn and the contents of the frame buffer. The following logical operations are supported on the PS/2:

1 AND
3 REPLACE
6 Exclusive-or
7 OR

You cannot change the cursor attributes while the cursor is visible.

There is no default cursor defined, so all cursor parameters must be set before the cursor is displayed.

For Pascal, the application must declare the array passed as being fixed length and declare the routine as accepting arrays of that length. The **k** in the routine declaration must be a constant.

Return Value

GS_SUCC Successful.
GS_COLI Invalid color index.
GS_CURS Cursor size invalid.
GS_CURO Cursor origin invalid.
GS_CURV Cursor visible.
GS_LONS Invalid logical operation.

Related Information
In this book: "gscatt" in topic 2.6.7, "gsecur" in topic 2.6.17, and "gsmcur" in topic 2.6.39.

2.6.39 gsmcur

Purpose
Moves the cursor and makes it visible.

C Syntax

int gsmcur_ (x, y)

int *x, *y;

FORTRAN Syntax

INTEGER function gsmcur (x, y)

INTEGER x, y

Pascal Syntax

FUNCTION gsmcur_ (

VAR x, y: INTEGER
): INTEGER [**PUBLIC];**

Description
The **gsmcur** subroutine makes the cursor visible (if not already visible) and positions the cursor origin at the point indicated by the parameters. This subroutine operates on either the single-color cursor or on the multicolor cursor. The relevant attributes are different, depending on which cursor style is currently defined.

For the single-color cursor, the relevant attributes are:

- Color ma
- Plane mas
- Cursor patter
- Cursor color inde
- Cursor origin

For the multicolor cursor, the relevant attributes are:

- Color ma
- Plane mas
- Multicolor cursor patter
- Multicolor cursor mas
- Multicolor cursor foreground colo
- Multicolor cursor background colo
- Multicolor cursor origi
- Multicolor cursor logical operation

Subtopics
2.6.39.1 Parameters

2.6.39.1 Parameters

x, y	Indicate the coordinates of the desired position of the cursor origin.

The cursor attributes must be set with the **gscatt** or **gsmcat** subroutine before calling **gsmcur**.

The cursor is non-destructive. This is achieved in a device-dependent manner.

Return Value

GS_SUCC	Successful.
GS_CORD	Invalid coordinate.
GS_UCUR	Undefined cursor.
GS_INAC	Virtual terminal inactive.

Related Information
In this book: "gscatt" in topic 2.6.7, "gsecur" in topic 2.6.17, and "gsmcat" in topic 2.6.38.

2.6.40 gsmult

Purpose
Draws a multiline, a set of lines that connect alternate pairs of points in a sequence.

C Syntax

```
int gsmult_ (number, x, y)

int *number, *x, *y;
```

FORTRAN Syntax

```
INTEGER function gsmult (number, x, y)

INTEGER number, x (*), y (*)
```

Pascal Syntax

```
FUNCTION gsmult_ (

VAR number: INTEGER;
VAR x, y: ARRAY [1..k] of INTEGER
): INTEGER [PUBLIC];
```

Description
The **gsmult** subroutine draws lines, as defined by the current relevant attributes, between alternate pair of points defined by the parameters.

The relevant attributes are:

- Color ma
- Plane mas
- Line color inde
- Line styl
- Logical operation

Subtopics
2.6.40.1 Parameters

2.6.40.1 Parameters

number	Defines the number of points in the coordinate arrays. It must be a multiple of 2, with 2 as the minimum value.
x, y	Define the points for line drawing.

For Pascal, the application must declare the arrays passed as being fixed length and declare the routine as accepting arrays of that length. The **k** in the routine declaration must be a constant.

Return Value

GS_SUCC	Successful.
GS_CORD	Invalid coordinate.
GS_NCOR	Invalid number of coordinates.
GS_INAC	Virtual terminal inactive.

2.6.41 gspcls

Purpose
Defines the end of a shape to fill.

C Syntax

int gspcls_ ( )

FORTRAN Syntax

INTEGER function gspcls

Pascal Syntax

FUNCTION gspcls_ : INTEGER [**PUBLIC];**

Description

The **gspcls** subroutine defines the end of a particular two dimensional shape to be filled, then fills the shape.

The relevant attributes are:

    Color ma
    Plane mas
    Fill color inde
    Fill styl
    Logical operation

Return Value

GS_SUCC Successful.
GS_USUC Unsuccessful.

Related Information
In this book: "gsbply" in topic 2.6.5 and "gseply" in topic 2.6.19.

2.6.42 gsplym

Purpose
Draws a polymarker, a marker at each of a set of specified points.

C Syntax

```
int gsplym_ (number, x, y)

int *number, *x, *y;
```

FORTRAN Syntax

```
INTEGER function gsplym (number, x, y)

INTEGER number, x (*), y (*)
```

Pascal Syntax

```
FUNCTION gsplym_ (

VAR number: INTEGER;
VAR x, y: ARRAY [1..k] of INTEGER
): INTEGER [PUBLIC];
```

Description
The **gsplym** subroutine places a marker, defined by the current relevant attributes, at each point defined by the parameters.

The relevant attributes are:

- Color ma
- Plane mas
- Logical operatio
- Polymarker color inde
- Polymarker style index

Subtopics
2.6.42.1 Parameters

2.6.42.1 Parameters

number	Defines the number of points in the coordinate arrays. It must be = 1.
x, y	Defines, as coordinate arrays, the location where the origin of each polymarker is placed.

For Pascal, the application must declare the arrays passed as being fixed length and declare the routine as accepting arrays of that length. The **k** in the routine declaration must be a constant.

Return Value

GS_SUCC	Successful.
GS_CORD	Invalid coordinate.
GS_NCOR	Invalid number of coordinates.

2.6.43 gspoly

Purpose
Draws a polyline, a set of lines that connects a sequence of points.

C Syntax

```
int gspoly_ (number, x, y)

int *number, *x, *y;
```

FORTRAN Syntax

```
INTEGER function gspoly (number, x, y)

INTEGER number, x (*), y (*)
```

Pascal Syntax

```
FUNCTION gspoly_ (

VAR number: INTEGER;
VAR x, y: ARRAY [1..k] of INTEGER
): INTEGER [PUBLIC];
```

Description
The **gspoly** subroutine draws lines, as defined by the current relevant attributes, between each pair of points defined by the parameters.

The relevant attributes are:

- Color ma
- Plane mas
- Line color inde
- Line styl
- Logical operation

Subtopics
2.6.43.1 Parameters

2.6.43.1 Parameters

number	Defines the number of points in the coordinate arrays. It must be = 2.
x, y	Defines the points for line drawing.

For Pascal, the application must declare the arrays passed as being fixed length and declare the routine as accepting arrays of that length. The **k** in the routine declaration must be a constant.

Return Value

GS_SUCC	Successful.
GS_CORD	Invalid coordinate.
GS_NCOR	Invalid number of coordinates.
GS_INAC	Virtual terminal inactive.

2.6.44 gspp

Purpose
Sets plotter pen speed.

C Syntax

```
int gspp_ (penspd)

int *penspd;
```

FORTRAN Syntax

```
INTEGER function gspp (penspd)

INTEGER penspd
```

Pascal Syntax

```
FUNCTION gspp_ (

VAR penspd: INTEGER;
): INTEGER [PUBLIC];
```

Description
The **gspp** subroutine sets the plotter pen speed.

Subtopics
2.6.44.1 Parameter

2.6.44.1 Parameter

penspd	Specifies the pen speed as a value from 0 to 100, giving a percentage of the maximum speed of the plotter. The initial pen speed is 100 percent.

Return Value

GS_SUCC	Successful.
GS_USUC	Invalid parameter value.

2.6.45 gsqdsp

Purpose
Returns characteristics of the display monitor and adapter.

C Syntax

```
void gsqdsp_ (display)

int *display;
```

FORTRAN Syntax

```
subroutine gsqdsp (display)

INTEGER display (32)
```

Pascal Syntax

```
PROCEDURE gsqdsp_ (

VAR display: ARRAY [1..32] of INTEGER) [PUBLIC];
```

Description
The **gsqdsp** subroutine returns an array containing the display adapter and monitor characteristics.

Subtopics
2.6.45.1 Parameter

2.6.45.1 Parameter

display Contains, on return, the relevant display/monitor characteristics. The following table describes the information in the array. Each entry is a word.

Entry	Description (measure)
1	Display/monitor ID. For a printer or plotter, this value is "**HC**", right-justified in the word.
2	Displayed width of the frame buffer in pixels.
3	Displayed height of the frame buffer in pixels.
4	Physical width of display in millimeters.
5	Physical height of display in millimeters.
6	Number of bit planes or number of bits/pixel.
7	Adapter characteristic flags. (Bit 0 is the most significant bit.) Bits set these characteristics: 0 Color or monochrome; 0 = color, 1 = monochrome 1 By plane or by pixel; 0 = by plane, 1= by pixel (always 1 for printers and plotters). 2 Software or hardware cursor; 0 = software, 1 = hardware (always 0 for printers and plotters). 3-31 Reserved bits.
8	Number of bits for Red digital-to-analog converter (always 2 for printers and plotters).
9	Number of bits for Green digital-to-analog converter (always 2 for printers and plotters).
10	Number of bits for Blue digital-to-analog converter (always 2 for printers and plotters).
11	Minimum cursor width (pixels) (always 0 for printers and plotters).
12	Minimum cursor height (pixels) (always 0 for printers and plotters).
13	Maximum cursor width (pixels) (always 0 for printers and plotters).
14	Maximum cursor height (pixels) (always 0 for printers and plotters).
15	Color table size. For printers and plotters, this specifies the number of colors.
16	Font class: 1 Compressed (always 1 for printers and plotters). 2 Uncompressed.
17	Logical operation capability. If the value is 0, the adapter supports all 16 two-operand logical operations and all 256 three-operand logical operations. If nonzero, the most significant bits represent the two-operand logical operations supported; bit 0 corresponds to logical operation 0, bit 1 to logical operation 1, and so on (see "gslop" in topic 2.6.35).
18-32	Reserved.

Information from this query can be used to scale application coordinates to those of the frame buffer.

Even if the adapter supports no logical operations, the results of the query indicate that the adapter supports REPLACE and Exclusive-or (logical operations 3 and 6, respectively). The GSL emulates the latter, if

necessary.

Related Information
In this book: "hft" in topic 2.5.11 and "gslop" in topic 2.6.35.

2.6.46 gsqfnt

Purpose
Returns information about the current font.

C Syntax

```
void gsqfnt_ (font)

int *font;
```

FORTRAN Syntax

```
subroutine gsqfnt (font)

INTEGER font (32)
```

Pascal Syntax

```
PROCEDURE gsqfnt_ (

VAR font: ARRAY [1..32] of INTEGER
) [PUBLIC];
```

Description
The **gsqfnt** subroutine returns information about the active font.

Subtopics
2.6.46.1 Parameter

2.6.46.1 Parameter

font Contains, on return, the characteristics of the current font. The following table describes the information in the array. Each entry is a word. Dimensions are in pixels and the origin is at the lower left corner of the character box.

Entry	Description
1	Class: 1 = compressed; 2 = uncompressed format (always 1 for printers and plotters).
2	Font ID.
3	Style.
4	Attribute flags: bit 31 bold bit 30 italic bit 00 proportionally spaced. (This entry always has all bits set to 0 for printers and plotters.)
5	Number of characters. For printers and plotters, this is the number of fonts ¦ 128.
6	Character baseline. For printers and plotters, no text alignment is allowed and this value is always -1.
7	Character capsline. For printers and plotters, no text alignment is allowed and this value is always -1.
8	Character width. For printers and plotters, the character width is given in pixels. For a proportionally spaced font, the width value represents the maximum width allowed.
9	Character height. For printers and plotters, the character height is given in pixels.
10	Underscore top line. For printers and plotters, underscoring is not available and this value is always -1.
11	Underscore bottom line. For printers and plotters, underscoring is not available and this value is always -1.
12-32	Reserved.

2.6.47 gsqgtx

Purpose
Returns information about the current geometric font.

C Syntax

```
void gsqgtx_ (font, select)

int *font, *select;
```

FORTRAN Syntax

```
subroutine gsqgtx (font, select)

INTEGER font (32), select
```

Pascal Syntax

```
PROCEDURE gsqgtx_ (

VAR font: ARRAY [1..32] of INTEGER;
select: INTEGER
) [PUBLIC];
```

Description
The **gsqgtx** subroutine returns information about the active geometric font.

Subtopics
2.6.47.1 Parameters

2.6.47.1 Parameters

font Contains, on return, the characteristics of the selected PCS descriptor header. The following table describes the information in the array. Each entry is a word. Dimensions are in pixels and the origin is at the lower left corner of the character box.

Entry	Description
1	Font ID.
2	Segment ID.
3	0 = EBCDIC; 1 = ASCII.
4	Range of **x** (P).
5	Range of **y** (Q).
6	Starting character code. Range is 0x21 to 0xFE.
7	Last character code. Range is 0x21 to 0xFE.
8	Font baseline. Value in pixels in the **y** direction.
9	Font capline. Value in pixels in the **x** direction.
10	Default error code point.
11-32	Reserved.

select Determines the type of query.

A value of -1 returns the following information in the **font** parameter buffer:

Word 1 Current active **font_ID**

Word 2 Number of PCS descriptor headers (segments for 2-byte text) loaded at the time of the query.

A value other than -1 returns the PCS descriptor header associated with that number in the table. The first entry is 0, with a range of 0 to **n**.

Related Information
In this book: "fonts" in topic 2.3.19, "Geometric Text Font Format" in topic 2.3.19.2, "gsgtat" in topic 2.6.28, and "gsgtxt" in topic 2.6.29.

2.6.48 gsqlext

Purpose
Returns expanded information about the locator.

C Syntax

```
int gsqlext_ (results )

int results[16];
```

FORTRAN Syntax

```
INTEGER function gsqlext (results)

INTEGER results (16 )
```

Pascal Syntax

```
FUNCTION gsqlext_ (

VAR results: ARRAY[1..16] of INTEGER;
): INTEGER [PUBLIC];
```

Description
The **gsqlext** subroutine returns an array containing expanded information about the locator device.

Subtopics
2.6.48.1 Parameter

2.6.48.1 Parameter

results Contains, on return, information about the type of locator, the resolution of the locator device, the maximum horizontal and vertical counts, and the current setting of the relative device thresholds or the absolute device dead zone values. The following table describes the information in the array.

Entry	Description
0	Locator resolution in millimeters per 100 counts.
1	Indicates the locator device type. If the most significant bit is: 0 The locator type is a mouse. When the locator is a mouse, the setting of the following bits is ignored. 1 The locator type is a tablet. For a tablet, the next most significant 2 bits are: 00 Sensor type is undefined or no sensor is attached. 01 A stylus is attached. 10 A four-button puck is attached.
2	Maximum horizontal count.
3	Maximum vertical count.
4	The horizontal locator threshold or dead zone in units of 0.25 millimeter.
5	The vertical locator threshold or dead zone in units of 0.25 millimeter.
6-15	Reserved.

An attempt to get the locator attributes can fail for a variety of reasons, the most likely of which is that the device is not attached. The nature of the problem can be found via a specific **ioctl** to the virtual terminal. (See "hft" in topic 2.5.11 for more information.)

For Pascal, the application must declare the array passed as being fixed length and declare the routine as accepting an array of that length. The **k** in the routine declaration must be a constant.

Return Value

GS_SUCC Successful.
GS_USUC Unsuccessful.

Related Information
In this book: "hft" in topic 2.5.11 and "gsqloc" in topic 2.6.49.

2.6.49 gsqloc

Purpose
Returns information about the locator.

C Syntax

```
void gsqloc_ (loc_type, x_res, y_res, hg, vg)

int *loc_type, *x_res, *y_res, *hg, *vg;
```

FORTRAN Syntax

```
subroutine gsqloc (loc_type, x_res, y_res, hg, vg)

INTEGER loc_type, x_res, y_res, hg, vg
```

Pascal Syntax

```
PROCEDURE gsqloc_ (

VAR loc_type, x_res, y_res, hg, vg: INTEGER
): INTEGER [PUBLIC];
```

Description
The **gsqloc** subroutine returns the type of the locator, the resolution of the device, and the current setting of the relative device thresholds or the absolute device dead zone values.

Subtopics
2.6.49.1 Parameters

2.6.49.1 Parameters

loc_type Indicates the type of locator. If the most significant bit of **loc_type** is 0, the locator is a mouse. When the locator is a mouse, the setting of the following bits is ignored. When the most significant bit is 1, the locator type is a tablet. For a tablet, the next most significant 2 bits are:

00 Sensor type is undefined or no sensor is attached.
01 A stylus is attached.
10 A four-button puck is attached.

x_res, y_res Indicate the horizontal and vertical resolution of the device in millimeters per 100 counts.

hg, vg Define the horizontal and vertical values for the locator threshold or dead zone in units of 0.25 millimeters.

An attempt to get the locator attributes can fail for a variety of reasons, the most likely of which is that the device is not attached. The nature of the problem can be found via a specific **ioctl** to the virtual terminal. (See "hft" in topic 2.5.11 for more information.)

Return Value

GS_SUCC Successful.
GS_USUS Unsuccessful.

Related Information
In this book: "hft" in topic 2.5.11 and "gslcat" in topic 2.6.32.

2.6.50 gsrrst

Purpose
Restores a rectangular block.

C Syntax

```
int gsrrst_ (buffer, x1,
y1, x2, y2)

int *buffer, *x1, *y1, *x2, *y2;
```

FORTRAN Syntax

```
INTEGER function gsrrst (buffer, x1, y1, x2, y2)

INTEGER buffer (*), x1, y1, x2, y2
```

Pascal Syntax

```
FUNCTION gsrrst_ (

VAR buffer: ARRAY [1..k] of INTEGER;
VAR x1, y1, x2, y2: INTEGER
): INTEGER [PUBLIC];
```

Description
The **gsrrst** subroutine restores a block of pixels saved to the frame buffer by the **gsrsav** subroutine.

The relevant attributes are:

    Plane mas
    Logical operation

Subtopics
2.6.50.1 Parameters

2.6.50.1 Parameters

buffer	Indicates where **gsrrst** should restore the block of pixels from. This stored block of pixels is typically a **buffer** saved in the **gsrsav** subroutine.
x1, y1	Define the coordinates of the lower left corner of the rectangular area to restore.
x2, y2	Define the coordinates of the upper-right corner of the rectangular area to restore.

The intended purpose of the **gsrsav** and **gsrrst** subroutines is efficient saving and restoring of pixel blocks displayed temporarily at a fixed location in the frame buffer. Because the GSL saves the frame buffer contents in a device-dependent fashion, it is generally not possible to use **gsrsav** and **gsrrst** to correctly move blocks of pixels from one position to another in a plane oriented adapter, nor is it possible for the application to manipulate the **buffer** without careful consideration of adapter characteristics, block size, and position of the block in the frame buffer.

For further information on moving and storing blocks of pixels, see "gsxblt" in topic 2.6.57.

For Pascal, the application must declare the array passed as being fixed length and declare the routine as accepting an array of that length. The **k** in the routine declaration must be a constant.

Return Value

GS_SUCC Successful.
GS_CORD Invalid coordinate.
GS_INAC Virtual terminal inactive.

Related Information
In this book: "gsrsav" in topic 2.6.51 and "gsxblt" in topic 2.6.57.

2.6.51 gsrsav

Purpose
Saves a rectangular block.

C Syntax

```
int gsrsav_ (buffer, x1, y1, x2, y2)

int *buffer, *x1, *y1, *x2, *y2;
```

FORTRAN Syntax

```
INTEGER function gsrsav (buffer, x1, y1, x2, y2)

INTEGER buffer (*), x1, y1, x2, y2
```

Pascal Syntax

```
FUNCTION gsrsav_ (

VAR buffer: ARRAY [1..k] of INTEGER;
VAR x1, y1, x2, y2: INTEGER
): INTEGER [PUBLIC];
```

Description
The **gsrsav** subroutine saves a block of pixels, defined by the input rectangle, in storage starting at the address indicated. This stored block can be restored with the **gsrrst** subroutine.

The relevant attributes are:

Plane mas
Logical operation

Subtopics
2.6.51.1 Parameters

2.6.51.1 Parameters

buffer Indicates where **gsrsav** should save the block of pixels.

The size of the **buffer** depends on the size of the rectangle and on the device organization. For devices organized by plane, the plane mask attribute determines the number of planes saved for each pixel. For devices organized by pixel, the entire pixel is always saved. For both organizations, the unit of access to the frame buffer also plays a role in calculating the size of the **buffer.** See "gscmap" in topic 2.6.11 for details.

Note that the **gsrsav** subroutine does not check whether the **buffer** is too small to contain the pixel block. Serious consequences can result if the **buffer** is too small. However, a **buffer** size equal to

```
((y2-y1+1) * ((x2-x1+1) / 32+2 ))
```

integers per plane will hold all save images on planar devices. For pixel devices, a **buffer** of

```
(y2-y1+1) * (x2-x1+1)/4 +1
```

integers is sufficient.

x1, y1 Define the lower left corner of the rectangular area to save. That is, **x1** is the greatest lower bound of the pixels saved in **x**.

x2, y2 Define the upper-right corner of the rectangular area to save. That is, **x2** is the least upper bound of the pixels saved in **x**.

The intended purpose of the **gsrsav** and **gsrrst** subroutines is efficient saving and restoring of pixel blocks displayed temporarily at a fixed location in the frame buffer. Because the GSL saves the frame buffer contents in a device-dependent fashion, it is generally not possible to correctly move blocks of pixels from one position to another in a plane-oriented adapter using **gsrsav** and **gsrrst**, nor is it possible to manipulate the **buffer** without careful consideration of adapter characteristics, block size, and position of the block in the frame buffer.

For further information on moving and storing blocks of pixels, see "gsxblt" in topic 2.6.57.

For Pascal, the application must declare the array passed as being fixed length and declare the routine as accepting an array of that length. The **k** in the routine declaration must be a constant.

Return Value

GS_SUCC Successful.
GS_CORD Invalid coordinate.
GS_INAC Virtual terminal inactive.

Related Information
In this book: "gscmap" in topic 2.6.11, "gsrrst" in topic 2.6.50, and

"gsxblt" in topic 2.6.57.

2.6.52 gstatt

Purpose
Sets the text attributes for annotated text.

C Syntax

```
int gstatt_ (color, page, baseline, font, name)

int *color, *page, *baseline, *font;
char *name;
```

FORTRAN Syntax

```
INTEGER function gstatt (color, page, baseline, font, name)

INTEGER color, page, baseline, font
CHARACTER*n name
```

Pascal Syntax

```
FUNCTION gstatt_ (

VAR color, page, baseline, font: INTEGER;
VAR name: ARRAY [0..k] of CHAR
): INTEGER [PUBLIC];
```

Description
The **gstatt** subroutine defines the attributes for the class of text drawing functions.

Subtopics
2.6.52.1 Parameters

2.6.52.1 Parameters

color Specifies a text color entry in the color map. If it is -1, the attribute is unchanged.

page Specifies the code page of a font for the display to use. The valid values for IBM supplied fonts are 0, 1, and 2 for code pages P0, P1, and P2, respectively. The value -1 indicates no change.

For printers and plotters, the **page** parameter is a font value specification. Again, the value -1 indicates no change.

baseline Determines the direction of the text drawing. The valid values are:

-1 Attribute remains unchanged.

0 Specifies 0 degrees, or left to right in the viewer's terms.

1 Specifies 90 degrees, or up in the viewer's terms.

2 Specifies 180 degrees, or right to left in the viewer's terms.

Note: The characters appear upside down.

3 For 270 degrees, or down in the viewer's terms.

If the baseline is other than 0 degrees and the font index is 0, then the font named by the **name** parameter must be a pre-rotated font. When a baseline change is made, another font path name is required.

font Specifies, for displays, the font to use for text output operations. For printers and plotters, the **font** parameter specifies the vertical height of the font in pixels.

If the **font** index is -1, no change is made. If the **font** index is 0, then **gstatt** uses the font specified by the **name** parameter. If the **font** index is a value from 1 to 14, the GSL uses one of the following predefined fonts:

	Font Index	**Width x Height (in pixels)**	**Style**	**Filename**
	1	9 x 20	Normal	nrm1.9x20
	2	9 x 20	Italic	itl1.9x20
	3	9 x 20	Bold	bld1.9x20
	4	8 x 14	Normal	nrm1.8x14

	5	4 x 8	Normal	nrm1.4x8
	6	18 x 40	Normal	nrm1.18x40
	7	12 x 30	Normal	nrm1.12x30
	8	9 x 20	Ergonomic	erg1.9x20
	9	6 x 9	Normal	nrm1.6x9
	10	6 x 11	Normal	nrm1.6x11
	11	7 x 15	Normal	nrm1.7x15
	12	7 x 22	Normal	nrm1.7x22
	13	11 x 23	Normal	nrm1.11x23
	14	11 x 23	Bold	bld1.11x23

All annotated text fonts are stored in the **/usr/lpp/gsl/fonts** directory. Many of the fonts are supplied with rotated versions.

name Contains the null-terminated full path name of the file used when the font attribute is specified as user. See "fonts" in topic 2.3.19 for the format of this file.

If a single-shift control is outstanding and **gstatt** is called to change the code page or the font, then the single-shift control is ignored.

For Pascal, the application must declare the arrays passed as being fixed length and declare the routine as accepting arrays of that length. The **k** in the routine declaration must be a constant.

Return Value

GS_SUCC Successful.
GS_COLI Invalid color index.
GS_CPID Invalid code page identifier.
GS_BASL Invalid baseline direction.
GS_FNTI Invalid font index.
GS_FNTN Invalid file name.

Related Information
In this book: "fonts" in topic 2.3.19.

2.6.53 gsterm

Purpose
Terminates use of the GSL.

C Syntax

void gsterm_ ( )

FORTRAN Syntax

subroutine gsterm ( )

Pascal Syntax

PROCEDURE gsterm_ ( ) [**PUBLIC];**

Description
The **gsterm** subroutine terminates the GSL. It deallocates any private storage required, returns the virtual terminal to KSR Mode, and causes the Monitor Mode signals to be ignored.

2.6.54 gstext

Purpose
Writes annotated text.

C Syntax

```
int gstext_ (x, y, number, text)

int *x, *y, *number;
char *text;
```

FORTRAN Syntax

```
INTEGER function gstext (x, y, number, text)

INTEGER x, y, number
CHARACTER*n text
```

Pascal Syntax

```
FUNCTION gstext_ (

VAR x, y, number: INTEGER;
VAR text: ARRAY [1..k] of CHAR
): INTEGER [PUBLIC];
```

Description
The **gstext** subroutine writes the number of characters indicated by the parameters, starting at the specified baseline position and according to the relevant attributes. This subroutine is to be used only with annotated text.

The relevant attributes are:

- Color ma
- Plane mas
- Fon
- Code pag
- Baseline directio
- Text color index

Subtopics
2.6.54.1 Parameters

2.6.54.1 Parameters

x, y	Define the baseline position for writing the text.
number	Indicates the number of bytes to write from the **text** string.
text	Contains the ASCII codes for the characters to write, as an array.

The graphics written to the frame buffer are determined by the 8-bit ASCII codes in the input data and the code page attribute. The ASCII control codes in between are ignored except the following: 1F, 1E, 1D, and 1C (hexadecimal). These control codes cause a shift to a predefined code page for the next ASCII character only.

The code page definitions are:

1F	Bottom half of code page 1
1E	Top half of code page 1
1D	Bottom half of code page 2
1C	Top half of code page 2.

For any ASCII value between 0 and 31 (decimal), no graphic is written. For any other ASCII value and code page combination that does not result in a valid graphic, a dash is written.

For Pascal, the application must declare the array passed as being fixed length and declare the routine as accepting an array of that length; the **k** in the routine declaration must be a constant.

Return Value

GS_SUCC	Successful.
GS_CORD	Invalid coordinate.
GS_FBUF	Frame buffer overflow.
GS_INAC	Virtual terminal inactive.

2.6.55 gsulns

Purpose
Sets the user line pattern.

C Syntax

int gsulns_ (pattern, length, begin)

int *pattern, *length, *begin;

FORTRAN Syntax

INTEGER function gsulns (pattern, length, begin)

INTEGER pattern, length, begin

Pascal Syntax

FUNCTION gsulns_ (

VAR pattern, length, begin: INTEGER
): INTEGER [**PUBLIC];**

Description
The **gsulns** subroutine establishes the user line style.

Subtopics
2.6.55.1 Parameters

2.6.55.1 Parameters

pattern Defines the pixel pattern used for the line style. A 1 bit indicates that the GSL draws a pixel; a 0 bit means that it does not.

length Defines the number of bits (starting with the most significant) of **pattern** used for line drawing. The bits are repeated for the length of the line.

The **length** parameter is a value not less than 2 or greater than 32.

begin Indicates the length of the starting run of bits set to 1 in the pattern. It is used to adjust the beginning and ending runs of the non-continuous line styles.

For proper appearance, the application-supplied line pattern should begin with a run of bits set to 1 and end with a run of bits set to 0.

Return Value

GS_SUCC Successful.
GS_ULNS Invalid user line style.

2.6.56 gsunlk

Purpose
Resumes signal processing.

C Syntax

void gsunlk_ ( )

FORTRAN Syntax

subroutine gsunlk ( )

Pascal Syntax

PROCEDURE gsunlk_ ( ) [**PUBLIC];**

Description
The **gsunlk** subroutine indicates to the GSL that the application is finished with the display adapter and it can now read the **SIGRETRACT** signal.

The application supplied routine called at **SIGRETRACT** can be entered as a result of **gsunlk**.

2.6.57 gsxblt

Purpose
Moves a rectangular block in system or display adapter memory from one location to another.

C Syntax

```
int gsxblt_ (srcpix, dstpix, mskpix, W, H, logop)

int *srcpix, *dstpix, *mskpix, *W, *H, *logop;
```

FORTRAN Syntax

```
INTEGER function gsxblt (srcpix, dstpix, mskpix, W, H, logop)

INTEGER srcpix(*), dstpix(*), mskpix(*), W, H, logop
```

Pascal Syntax

```
FUNCTION gsxblt_ (

VAR srcpix, dstpix, mskpix: ARRAY [32] of INTEGER;
VAR W, H, logop : INTEGER
): INTEGER [PUBLIC];
```

Description

The **gsxblt** subroutine moves a rectangular block of pixels from one memory location to another, either in system memory or in the display adapter frame buffer.

The **gsxblt** subroutine is used to support windowing operations, such as overlays and movement around the screen. The source rectangle and the destination rectangle can be in either system or adapter pixel memory. The **gsxblt** subroutine is also used for user defined cursors and the save and restore of a pixel map for applications like pop-up menus.

The mask operation provided by the **gsxblt** subroutine controls which pixels in the destination rectangle can be modified.

The relevant attributes are:

- Plane mas
- Color map

Subtopics
2.6.57.1 Parameters

2.6.57.1 Parameters

srcpix Contains the address of the source pixel map.

dstpix Contains the address of the destination pixel map.

mskpix Contains the address of the mask operation pixel map. This parameter should equal 0 if there is no bit mask operator to apply. For Fortran applications, a valid **mskpix** array must always be defined. If no masking is required, the address field of the array, **mskpix**[9], must be initialized to 0.

The **mskpix** pixel map must always consist of only 1 bit per pixel, and the mask rectangle must always be the same size as the source and destination rectangles. In the mask rectangle, a 1 bit means that the corresponding pixel in the destination rectangle can be modified, while a 0 bit means the destination pixel will not be modified.

W Defines the width of the rectangular area to be transferred.

H Defines the height of the rectangular area to be transferred.

logop Indicates the logical operation to perform between the source pixel map and the destination pixel map.

In the following table, please note:

The source or tile (a special type of source) pixels represent bits of data to be merged in some way with the corresponding bits of data in the destination rectangle.

The first three columns of the table specify the operations you can perform, and the **Code** column contains the corresponding value you should specify for the **logop** parameter.

There are two unique codes for each logical operation, to be used depending on whether the tiling bit in the source pixel map is set. Codes 0-15 must be used when the tiling bit is not set, while codes 16-31 must be used when the tiling bit is set.

A ~ (tilde) represents the logical INVERSE.

Type of Source	**Logical Operation**	**Type of Destination**	**Co**
		Destination clear	0
		Set Destination	15
	No operation	Destination	5
		~Destination	10
Source	REPLACE	Destination	3

Source	AND	Destination	1
Source	AND	~Destination	2
Source	Exclusive-or	Destination	6
Source	OR	Destination	7
Source	OR	~Destination	11
~Source	REPLACE	Destination	12
~Source	AND	Destination	4
~Source	AND	~Destination	8
~Source	Exclusive-or	Destination	9
~Source	OR	Destination	13
~Source	OR	~Destination	14
		Destination clear	16
		Set Destination	31
	No operation	Destination	21
		~Destination	26
Tile	REPLACE	Destination	19
Tile	AND	Destination	17
Tile	AND	~Destination	18
Tile	Exclusive-or	Destination	22
Tile	OR	Destination	23
Tile	OR	~Destination	27
~Tile	REPLACE	Destination	28
~Tile	AND	Destination	20
~Tile	AND	~Destination	24
~Tile	Exclusive-or	Destination	25
~Tile	OR	Destination	29
~Tile	OR	~Destination	30

A ***pixel map*** is a 32-bit array of integers that contains the following fields:

0 Device ID (0 for memory) (from **gsqdsp**)

1 Flags

In the following explanations, bit 0 is the low-order bit.

Plane (XY) format is selected when bit 0 is set and bits 1 and 2 are not set. Pixel (Z) format is selected when bits 0, 1, and 2 are not set.

A repetitive tile is specified when bit 3 is set, while no tile is specified when bit 3 is not set.

If the repetitive tile bit is set in the **srcpix**, pixel map, then the Device ID field in that pixel map must equal 0. The tile data must be in memory.

Bit 4 selects the lower-left coordinate system when it is set, and the upper-left coordinate system when it is not set.

2 Height (in pixels)

3 Width (in pixels)

This value must be an even multiple of 16 pixels for all pixel maps, which means that all pixel maps must be at least 16 pixels wide.

4 Number of bits per pixel

5 Pixels per byte, right justified

6 Bytes per pixel

7 x offset

8 y offset

9 Address of upper-left corner of data

10 Foreground color index

11 Background color index

12 - 31 Reserved.

Definitions of pixel map terms include:

Device ID
This is a required parameter for all pixel map definitions. If the pixel map being defined is a display adapter, this field must contain the Device ID of that display adapter. If the pixel map resides in system memory, then this field must equal 0.

Pixel format
Data stored in this format has all bits for a pixel stored together. The data starts with the origin and increases first in the **x** direction, then in the **y** direction.

As an example using the upper-left coordinate system, a pixel map with 4 bits per pixel and 1 pixel per byte stores the 4 bits for the pixel at location (0,0) in the first byte of the data area, right justified in the byte. The 4 bits for the pixel at location (1,0) are stored in the second byte, followed by the rest of the pixel values in that row. When the end of the row is reached, the next byte contains the 4 bits for the pixel at location (0,1), followed by the rest of the pixel values in that row, and so on for the entire image.

Plane format
Plane format indicates that each of the bits that make up a pixel is stored in a separate, consecutive plane in memory. The most significant bit is first, followed by the next significant, and so on to the least significant bit, which is last. The bits within a plane are packed together 8 bits per byte. Therefore, using the upper-left coordinate system as an example, a pixel map with 4 bits per pixel would consist of four separate planes of data with the first bit value being the one for location (0,0) and increasing first in the **x** direction, then in the **y** direction.

Repetitive tiling operation
This operation consists of repeatedly copying a 16-pixel wide by 16-pixel high tile rectangle pointed to by the tile pixel map data address to fill a rectangular area of a size specified by the **H** and **W** parameters of this call. The format of the tile data is determined by the format defined in the **flags** field of

the tile pixel map structure.

Upper-left coordinate system
This indicates that the upper-left corner of the pixel map is used as the origin of the coordinate system, with increasing values of **x** moving to the right and increasing values of **y** moving down. The **x offset** and **y offset** are to set the upper-left corner of the rectangle when using this coordinate system.

Lower-left coordinate system
This indicates that the lower-left corner of the pixel map is used as the origin of the coordinate system, with increasing values of **x** moving to the right and increasing values of **y** moving up. The **x offset** and the **y offset** are set to the lower-left corner of the rectangle when using this coordinate system. Note, however, that the **data address** specified in the pixel map structure must always point to the upper-left corner of the data area no matter which coordinate system is defined.

Number of bits per pixel
This field identifies the number of bits of data required to define a pixel value. For example, a simple monochrome display requires only 1 bit per pixel, while a color display may require 4 bits of information to define a pixel.

Number of pixels per byte
If the number of bits per pixel is less than 8, this field defines how many pixels are stored in each byte of pixel map data. A pixel map with only 1 bit per pixel must always store 8 pixels per byte. It is strongly recommended that for between 2 and 7 bits per pixel, you store data with only 1 pixel per byte.

Bytes per pixel
If the number of bits per pixel is greater than 8, this field defines how many bytes are used to store each pixel. It is strongly recommended that for between 9 and 16 bits per pixel, you store data 2 bytes per pixel. For between 17 and 32 bits per pixel, data should be stored 4 bytes per pixel.

Foreground color index
This specifies the color index value to use for a value of 1 in the source pixel map during a color expansion operation.

Background color index
This specifies the color index value to use for a value of 0 in the source pixel map during a color expansion operation.

A ***color expansion operation*** takes place automatically when the source pixel map data area contains only 1 bit per pixel and the destination pixel map data area is a color display adapter frame buffer defined to have more than 1 bit per pixel. In this case, when a 1 is specified in the source pixel map data area, the **foreground color index** value specified in the destination pixel map (**dstpix**) is written to the destination data area. When a 0 is specified in the source pixel map data area, the **background color index** value specified in the destination pixel map (**dstpix**) is written to the destination data area.

The **foreground color index** and the **background color index** must be initialized in the **dstpix** pixel map before calling this operation, but do

not need to be initialized in the **srcpix** or **mskpix** pixel maps.

For VGA, not all logical operations are supported for a color expansion operation. The following table shows which operations are supported. In this table, a ~ (tilde) represents the logical INVERSE. Note that the operations in the left column of the table are for source pixel maps, while the operations in the right column are for tile pixel maps.

Type of Operation	Code	Type of Operation	Code
Destination clear	0	Destination clear	16
Set destination	15	Set destination	31
Destination	5	Destination	21
~ Destination	10	~ Destination	26
Source	3	Tile	19
~ Source	12	~ Tile	28

If a source or destination pixel map structure defines the active display adapter, you do not need to initialize all the fields of that pixel map structure. Device-dependent information, such as **height**, **width**, **pixels per byte**, **bytes per pixel**, and **address of data**, is supplied automatically. You must initialize the fields for **device ID**, **bits per pixel**, **flags** (except for the data format bits), **x offset**, and **y offset**. Also, the **foreground color index** and the **background color index** must be initialized if appropriate for this adapter.

When initializing a pixel map structure to use as the **mskpix** parameter:

1. The **flags** field should equal a value of 0x01 if the upper-left coordinate system will be used or 0x11 if the lower left coordinate system will be used.
2. The number of **bits per pixel** must equal 1.
3. The number of **pixels per byte** must equal 8.

The GSL plane mask attribute applies to all **gsxblt** operations that use the display adapter as the source or destination pixel map.

The GSL color map attribute applies to all **gsxblt** operations that use the display adapter as the destination pixel map.

Return Value

GS_SUCC Successful.
GS_IWID Invalid width specification. The **x_offset** plus the **W** parameter of one of the pixel maps exceeds the total width of that pixel map.
GS_IHEI Invalid height specification. The **y_offset** plus the **H** parameter of one of the pixel maps exceeds the total height of that pixel map.
GS_NPLF Source and destination data formats do not match.
GS_INAC Virtual terminal inactive.
GS_CORD Invalid coordinate specified that placed the origin of the source, destination, or mask rectangle outside its pixel map.
GS_IBPP Invalid value for **bits per pixel** in the source pixel map.
GS_CEXP Color expansion operation attempted, but the destination pixel map was not a display adapter.
GS_PWID The width of one of the pixel maps is not an even multiple of 16 pixels.

Related Information
In this book: "gsxptr" in topic 2.6.59.

2.6.58 gsxcnv

Purpose
Converts pixel map data organization.

C Syntax

```
int gsxcnv_ (inppix, outpix)

int *inppix, *outpix;
```

FORTRAN Syntax

```
INTEGER function gsxcnv (inppix, outpix)

INTEGER inppix(*), outpix(*)
```

Pascal Syntax

```
FUNCTION gsxcnv_ (

VAR inppix, outpix: INTEGER
): INTEGER [PUBLIC];
```

Description

The **gsxcnv** subroutine converts pixel map data to and from planes. That is, **gsxcnv** converts XY form to and from pixels, or Z form.

Subtopics
2.6.58.1 Parameters

2.6.58.1 Parameters

inppix	Points to the address of the pixel map that contains the address of the data area to be converted.
outpix	Points to the address of the pixel map that contains the address of where to put the converted data.

Both the **inppix** and **outpix** parameters contain the address of a pixel map. The fields of each pixel map must be completely initialized before calling this subroutine. Both pixel maps must point to data areas that reside in system memory, not in a display adapter frame buffer.

The **inppix** and **outpix** pixel maps do not have to specify the same number of bits per pixel. If there are more input bits per pixel, the least significant bits are truncated. If there are less input bits per pixel than required to fill out the destination, the most significant bits are filled with zeros.

The **gsxcnv** subroutine only supports pixel maps defined to have 8 bits per pixel or less. If a pixel format pixel map is defined with less than 8 bits per pixel, the data must be arranged 1 byte per pixel, right justified in that byte.

The widths and heights of the two data areas must be identical.

Warning: The calling process must allocate enough storage in the area pointed to by the **outpix** pixel map to contain all of the converted data. For pixel-oriented data, a **buffer** of

```
height * width/4
```

integers is sufficient. For plane-oriented data, a **buffer** of

```
((height * width)/32+1) * bits_per_pix
```

integers is sufficient.

Return Value

GS_SUCC	Successful.
GS_INPF	Invalid data format specified in **inppix** pixel map structure.
GS_OUTF	Invalid data format specified in **outpix** pixel map structure.
GS_BMAX	Pixel map defines data of more than 8 bits per pixel.

Related Information
In this book: "gsxblt" in topic 2.6.57.

2.6.59 gsxptr

Purpose
Handles FORTRAN addressing of data.

C Syntax

None

FORTRAN Syntax

INTEGER function gsxptr (intptr, datptr)

INTEGER intptr(*), datptr(*)

Pascal Syntax

None

Description

The **gsxptr** subroutine places a data address in a variable so that the data address field of a pixel map structure can be initialized.

In a FORTRAN application, you must first call the **gsxptr** subroutine, then the **gsxblt** subroutine.

Subtopics
2.6.59.1 Parameters

2.6.59.1 Parameters

intptr	Contains the address of the variable containing the data area.
datptr	Will be initialized to the address of the data area.

Return Value

GS_SUCC Successful.

Related Information
In this book: "gsxblt" in topic 2.6.57.

2.6.60 gsxtat

Purpose
Sets the text attributes for annotated text using the **rtfont** format.

C Syntax

```
int gsxtat_ (foreground, background, logop, font, clipbox)

int *foreground, *background, *logop, *font, *clipbox;
```

FORTRAN Syntax

```
INTEGER function gsxtat (foreground, background, logop, font, clipbox)

INTEGER foreground, background, logop, font, clipbox
```

Pascal Syntax

```
FUNCTION gsxtat_ (

VAR foreground, background, logop, : INTEGER;
VAR font: ARRAY [1..k] of INTEGER;
VAR clipbox: ARRAY [1..l] of INTEGER;
): INTEGER [PUBLIC];
```

Description
The **gsxtat** subroutine defines the attributes to be used when drawing text with a font in the **rtfont** format.

Subtopics
2.6.60.1 Parameters

2.6.60.1 Parameters

foreground Defines an entry in the color map to use for the foreground color (bits set to 1) in the font raster for each character. A value of -1 indicates no change for this attribute.

background Defines an entry in the color map to use for the background color (bits set to 0) in the font raster for each character. A value of -1 indicates no change for this attribute.

logop Indicates the logical operation to perform between the font raster and the display destination.

In the following table, please note:

The source pixels represent bits of data from the font raster to be merged in some way with the corresponding bits of data in the destination rectangle.

The first three columns of the table specify the operations you can perform, and the **Code** column contains the corresponding value you should specify for the **logop** parameter.

A ~ (tilde) represents the logical INVERSE.

Type of Source	**Logical Operation**	**Type of Destination**	**Code**
		Destination clear	0
		Set Destination	15
	No operation	Destination	5
		~Destination	10
Source	REPLACE	Destination	3
Source	AND	Destination	1
Source	AND	~Destination	2
Source	Exclusive-or	Destination	6
Source	OR	Destination	7
Source	OR	~Destination	11
~Source	REPLACE	Destination	12
~Source	AND	Destination	4
~Source	AND	~Destination	8
~Source	Exclusive-or	Destination	9
~Source	OR	Destination	13
~Source	OR	~Destination	14

A value of -1 for this parameter indicates no change in the current logical operation.

font Points to a data area that contains the font header and raster definitions for all characters defined in the font. The calling process is responsible for either mapping the font file or copying it into a memory area in order to obtain a pointer to the data area.

Setting the value of this pointer to 0 indicates no

change to the current font file.

The GSL supports only a subset of the different forms that the **rtfont** format allows. Specifically, the GSL supports any combination of the following font formats:

fixed width and height

variable width and/or height

halfword alignment or fullword alignment

glyphs in raster format only

index character array width of 4 bytes

all individual glyph character bounds for variable width and height fonts, except negative left or right bearings.

For more information on valid formats for **rtfont** files, see "fonts" in topic 2.3.19 and the **rtfont.h** header file.

The GSL does not support any formats for **rtfont** files other than those listed above. If the font file specified is not in a supported format, then the GSL returns the **GS_FFMT** return code.

clipbox Specifies an array of integers that correspond to a rectangular area on the display screen. When the **gsxtxt** subroutine is used to draw text, any full or partial characters that fall outside this area are clipped. The elements of the area to clip are as follows:

first element	Reserved. This value should always be 1.
second element	Specifies the x coordinate of the lower left corner of the clip box, in pixels.
third element	Specifies the y coordinate of the lower left corner of the clip box, in pixels.
fourth element	Specifies the height of the clip box, in pixels.
fifth element	Specifies the width of the clip box, in pixels.

The bottom and left edges of the clip box are inclusive, while the top and right edges are exclusive.

This parameter is a pointer to the clip box array, which is not copied into any GSL data structure, allowing the calling process to modify the elements of the array without calling the **gsxtat** subroutine. If the values for the clip box are changed between calls to the **gsxtxt**

subroutine, the new clip box is used for all text drawing until another change is made.

Setting the value of this pointer to 0 indicates no change.

Warning: Since the GSL subroutines that use the **rtfont** format are designed for high-performance text drawing, no verification is made of the validity of the clip box. It is the responsibility of the calling process to ensure that the entire clip box resides inside the physical size of the display. Using a clip box that is not entirely within the screen will produce unpredictable results.

When the GSL is installed from diskette, an attempt is made to convert the 14 supplied VRM format fonts into **rtfont** format. The **vrm2rtfont** command (described in the *AIX Operating System Commands Reference*) is used on the 14 VRM fonts in the **/usr/lpp/gsl/fonts** directory, and the resulting **rtfont**s are stored in the **/usr/lpp/fonts** directory. The following list shows the **rtfont** format files stored in **/usr/lpp/fonts**:

Width x Height (in pixels)	**Style**	**Filename**
4 x 8	Normal	Rom6.500
6 x 9	Normal	Rom7.500
6 x 11	Normal	Rom8.500
7 x 15	Normal	Rom11.500
7 x 22	Normal	Rom16.500
8 x 14	Normal	Rom10.500
9 x 20	Normal	Rom14.500
9 x 20	Italic	Itl14.500
9 x 20	Bold	Bld14.500
9 x 20	Ergonomic	Erg14.500
11 x 23	Normal	Rom17.500
11 x 23	Bold	Bld17.500
12 x 30	Normal	Rom22.500
18 x 40	Normal	Rom29.500

All of these fonts have fixed width and height and are halfword aligned.

For Pascal, the application must declare the arrays passed as being fixed length and declare the routine as accepting arrays of that length. The **k** and **l** in the routine declaration must be constants.

Return Value

GS_SUCC Successful.
GS_FFMT Invalid font format.
GS_LONS Invalid logical operation.

File
/usr/include/rtfont.h

Related Information
In this book: "fonts" in topic 2.3.19 and "gsxtxt" in topic 2.6.61.

The **vrm2rtfont** command in *AIX Operating System Commands Reference*.

2.6.61 gsxtxt

Purpose
Writes annotated text using the **rtfont** format.

C Syntax

```
int gsxtxt_ (x, y, number, text)

int *x, *y, *number;
char *text;
```

FORTRAN Syntax

```
INTEGER function gsxtxt (x, y, number, text)

INTEGER x, y, number
CHARACTER*n text
```

Pascal Syntax

```
FUNCTION gsxtxt_ (

VAR x, y, number: INTEGER;
VAR text: ARRAY [1..k] of CHAR
): INTEGER [PUBLIC];
```

Description
The **gsxtxt** subroutine displays the specified text string using the **rtfont** format. Only those full or partial characters that fall within the clip box specified by the **gsxtat** subroutine are displayed. In addition, since there is no default **rtfont** defined for use by the GSL, the **gsxtat** subroutine must be called to set all relevant attributes before the first call to this subroutine.

The relevant attributes are:

- xtext foreground colo
- xtext background colo
- xtext logical operatio
- xtext clip bo
- xtext current **rtfont** file
- plane mas
- color map

Subtopics
2.6.61.1 Parameters

2.6.61.1 Parameters

x, y	Define the baseline position for writing the text.
number	Indicates the number of bytes to write from the **text** string.
text	Contains the ASCII codes for the characters to write, as an array.

For Pascal, the application must declare the array passed as being fixed length and declare the routine as accepting an array of that length; the **k** in the routine declaration must be a constant.

Return Value

GS_SUCC Successful.

Since the text is either displayed or clipped in any case, the **gsxtxt** subroutine always completes with a successful return code.

File
/usr/include/rtfont.h

Related Information
In this book: "fonts" in topic 2.3.19 and "gsxtat" in topic 2.6.60.

A.0 Appendix A. Error Codes

This section describes the error conditions that can occur when using the system calls described in this book. Some subroutines also use these codes to indicate errors.

System calls indicate the fact that an error has occurred by returning a special value. This value is almost always -1, but check the description of the individual system call to be sure. Also, a number that identifies the error is stored in an external variable named **errno**. The **errno** variable is not cleared when a system call finishes successfully, so its value is meaningful only after an error has occurred.

If you are going to check the value of **errno** in a program, include the following line at the top of the source file:

```
#include <errno.h>
```

The **errno.h** header file declares the **errno** variable and defines the name of each error condition.

For each error code, the following list shows the symbolic name defined in the **/usr/include/errno.h** header file, the corresponding numeric value, and a brief description of the error:

EPERM (1) **Operation not permitted**

Cause: You attempted to modify a file in some way forbidden except to the owner of the file or to superuser. Or, a user other than superuser attempted to do something that only superuser is allowed to do.

ENOENT (2) **No such file or directory**

Cause: The file specified does not exist, or one of the directories in a path name does not exist.

ESRCH (3) **No such process**

Cause: A process corresponding to that specified in the **pid** parameter of the **kill** or **ptrace** system calls cannot be found.

EINTR (4) **Interrupted system call**

Cause: An asynchronous signal (such as interrupt or quit), which you have elected to catch, occurred during a system call. If the signal handler performs a normal return, the interrupted system call may return this error condition.

EIO (5) **Input/Output error**

Cause: A physical I/O error occurred. In some cases, this error occurs on a system call following the one to which it actually applies.

ENXIO (6) **No such device or address**

Cause: I/O on a special file referred to a device or subdevice that does not exist or referred to an address that is beyond the limits of the device.

E2BIG (7) **Arg list too long**

Cause: The combined length of the argument list and the environment list passed to one of the **exec** system calls totaled more than 10,240 bytes.

ENOEXEC (8) **Exec format error**

Cause: A request was made to execute a file that has the appropriate permissions, but does not start with a valid text header. (For information about text headers, see "a.out" in topic 2.3.2.)

EBADF (9) **Bad file descriptor**

Cause: A file descriptor was specified that does not refer to an open file, or a read request was made to a file that is open only for writing, or a write request was made to a file that is open only for reading.

ECHILD (10) **No child processes**

Cause: A process that invoked the **wait** system call has no existing child processes that have not already been waited for.

EAGAIN (11) **Resource temporarily unavailable**

Cause: The **fork** system call failed because the system's process table is full or the user is not allowed to create any more processes. Or, an attempt was made to access a region of a file that has an outstanding enforcement-mode lock. (See "fcntl, flock, lockf" in topic 1.2.78 about file locking.)

ENOMEM (12) **Not enough space**

Cause: During a **brk**, **sbrk**, or **exec** system call, a program asked for more space than the system is able to supply. This is not a temporary condition. The maximum space size is a system parameter.

EACCES (13) **Permission denied**

Cause: An attempt was made to access a file in a way that is forbidden by the protection system.

EFAULT (14) **Bad address**

Cause: An address passed to a system call that points to a location outside of the process's allocated address space.

ENOTBLK (15) **Block device required**

Cause: A nonblock file was specified when a block device is required, such as in the **mount** system call.

EBUSY (16) **Resource busy**

Cause: An attempt was made to mount a device that is already mounted, or an attempt was made to dismount a device on which there is an active file. This error also occurs when an attempt is made to enable accounting when it has already been enabled.

EEXIST (17) **File exists**

Cause: A previously existing file was specified to a system call or subroutine that would replace that file, such as the **link** system call.

EXDEV (18) **Improper link**

Cause: An attempt was made to link to a file on another device. (See "link" in topic 1.2.156.)

ENODEV (19) **No such device**

Cause: An attempt was made to use an inappropriate system call to a device, for example, to write to a read-only device.

ENOTDIR (20) **Not a directory**

Cause: A nondirectory parameter was specified where a directory is required, for example in a path prefix or as a parameter to the **chdir** system call.

EISDIR (21) **Is a directory**

Cause: An attempt was made to write on a directory.

EINVAL (22) **Invalid argument**

Cause: An invalid argument or action was specified to a system call, such as dismounting a device that is not mounted, specifying an undefined signal, or writing to a file for which **lseek** has generated a negative file pointer.

ENFILE (23) **Too many open files in system**

Cause: An attempt was made to open a file, and the system's table of open files is full. The maximum number of open files is a system parameter in **/etc/system**.

EMFILE (24) **Too many open files**

Cause: A process attempted to open more than two hundred (200) file descriptors at one time.

ENOTTY (25) **Inappropriate I/O control operation**

Cause: An **ioctl** system call was issued to a special file that does not support **ioctl**.

ETXTBSY (26) **Text file busy**

Cause: This error occurs when an attempt is made to execute a pure-procedure program that is currently open for writing or reading. It also occurs when an attempt is made to open for writing or to remove a pure-procedure program or shared library while that program or library is being executed.

EFBIG (27) **File too large**

Cause: The size of a file exceeded the maximum file size (2,147,483,648 bytes), or the maximum size set by the **ulimit** system call.

ENOSPC (28) **No space left on device**

Cause: During a write to an ordinary file, the device ran out of free space on the file system where the file resides.

ESPIPE (29) **Invalid seek**

Cause: An **lseek** system call was issued to an unseekable file or device, such as a pipe.

EROFS (30) **Read-only file system**

Cause: An attempt was made to modify a file or directory on a device that is mounted as read-only.

EMLINK (31) **Too many links**

Cause: An attempt was made to make more than the maximum number of links (1000) to a file.

EPIPE (32) **Broken pipe**

Cause: An attempt was made to write to a pipe for which there is not a process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

EDOM (33) **Domain error**

Cause: A parameter to a Math Library (**libm.a**) subroutine was out of the domain of the function.

ERANGE (34) **Result too large**

Cause: The return value of a Math Library (**libm.a**) subroutine is not representable within machine precision.

ENOMSG (35) **No message of desired type**

Cause: An attempt was made to receive a message of a type that does not exist on the specified message queue.

EIDRM (36) **Identifier removed**

Cause: The specified identifier has been removed from the file system's name space. (See "msgctl" in topic 1.2.173, "semctl" in topic 1.2.243, and "shmctl" in topic 1.2.259.)

Note: The values **ECHRNG (37)** through **EL2HLT (44)** are supplied in the **errno.h** header file for compatibility with UNIX System V. These values are not set by any AIX software.

ECHRNG (37) **Channel number out of range**
EL2NSYNC (38) **Level 2 not synchronized**
EL3HLT (39) **Level 3 halted**
EL3RST (40) **Level 3 reset**
ELNRNG (41) **Link number out of range**
EUNATCH (42) **Protocol driver not attached**

ENOCSI (43) **No CSI structure available**
EL2HLT (44) **Level 2 halted**

EDEADLK (45) **Resource deadlock avoided**

Cause: A potential deadlock was detected while attempting to lock a region of a file with the **lockf** system call.

ENOTREADY (46) **Device not ready**

Cause: The device is not ready for operation. For example, a diskette drive does not contain a diskette, or the device is not powered on.

EWRPROTECT (47) **Write-protected media**

Cause: There was an attempt to write to a device whose I/O media is write-protected.

EFORMAT (48) **Unformatted or incompatible media**

Cause: There was an attempted I/O operation on a device with media which has not been formatted or the format is not compatible with the I/O device.

ENOLCK (49) **No locks available**

Cause: There are no more file locks available. Too many segments are already locked.

ENOCONNECT (50) **Cannot Establish Connection**

EBADCONNECT (51) **Connection Down**

ESTALE (52) **Missing file or file system**

Cause: Either the file system of a remote (NFS) file has been unmounted, or the file descriptor of a remote (NFS) file has become obsolete.

EDIST (53) **Requests blocked by Administrator**

EWOULDBLOCK (54) **Operation would block**

EINPROGRESS (55) **Operation now in progress**

Cause: The socket was marked **O_NDELAY** by an **fcntl** system call, then a **connect** operation was attempted that has not completed yet.

EALREADY (56) **Operation already in progress**

Cause: The requested socket connection or disconnection is already in progress.

ENOTSOCK (57) **Socket operation on non-socket**

Cause: The command cannot complete because the file descriptor specified is not a socket.

EDESTADDRREQ (58) **Destination address required**

Cause: The attempted socket operation failed because a destination address was required, but not provided.

EMSGSIZE (59) Message too long

Cause: The socket data transfer failed because the message exceeded the size limits.

EPROTOTYPE (60) Protocol wrong type for socket

Cause: Either the two sockets to be connected are not of the same type, or the protocol used does not support this type of socket.

ENOPROTOOPT (61) Protocol not available

Cause: The protocol specified either does not support this particular option or does not support any options.

EPROTONOSUPPORT (62) Protocol not supported

Cause: No protocol of the specified type and domain exists.

ESOCKTNOSUPPORT (63) Socket type not supported

Cause: The type of socket specified is not supported. Do not use this type of socket in your program.

EOPNOTSUPP (64) Operation not supported on socket

Cause: This socket, with its particular type, domain, and protocol, does not allow the requested operation.

EPFNOSUPPORT (65) Protocol family not supported

Cause: The socket protocol specified is not supported. Do not use this protocol in your program.

EAFNOSUPPORT (66) Address family not supported by protocol

Cause: The socket name is of a type that is not valid in this socket or for the domain.

EADDRINUSE (67) Address already in use

Cause: A **bind** or **connect** operation was attempted using a socket name that is already in use.

EADDRNOTAVAIL (68) Cannot assign requested address

Cause: The requested socket name is not available to this machine. Either an incorrect socket name was used, or there is a problem at the remote node where the socket name should be.

ENETDOWN (69) Network is down

Cause: A socket operation failed because the network is down.

ENETUNREACH (70) Network is unreachable

Cause: A socket operation failed because the destination is at a remote

node that cannot be reached over the network.

ENETRESET (71) Network dropped connection on reset

Cause: The host the socket was connected to went down. The connection can be reestablished after the remote node is restarted.

ECONNABORTED (72) Software caused connection abort

Cause: The connection between a socket and a remote node was terminated at the local node, the remote node, or the network level.

ECONNRESET (73) Connection reset by peer

Cause: The connection with another socket was reset by that socket. This **errno** can be set due to an error, or just due to a connection that was closed.

ENOBUFS (74) No buffer space available

Cause: Not enough buffer space is available for the requested socket operation.

EISCONN (75) Socket is already connected

Cause: A **connect** operation was attempted on a socket that is already connected.

ENOTCONN (76) Socket is not connected

Cause: A socket operation other than a **connect** was attempted on a socket that is not currently connected, or a **send** operation that does not require a connection was attempted without a destination address.

ESHUTDOWN (77) Cannot send after socket shutdown

Cause: An attempt was made to send data after a **shutdown** operation was done on the socket.

ETIMEDOUT (78) Connection timed out

Cause: A remote socket did not respond within the timeout period set by the protocol of the socket on this node.

ECONNREFUSED (79) Connection refused

Cause: A remote node refused to allow the attempted **connect** operation.

EHOSTDOWN (80) Host is down

Cause: A socket operation failed because the remote node specified is down.

EHOSTUNREACH (81) No route to host

Cause: A socket operation failed because no route to the remote node was available due to an incorrect address, an incorrect routing table, or network hardware problems.

EDUPLOOP (82) Reserved

ENOLOOP (83) **Reserved**

ENOTINLOOP (84) **Reserved**

ELOOP (85) **Symbolic link loop**

Cause: A path name lookup involved more than eight symbolic links.

ENAMETOOLONG (86) **File name too long**

Cause: A component of a path name exceeded 255 (**MAXNAMELEN**) characters or an entire path name exceeded 1023 (**MAXPATHLEN-1**).

ENOTEMPTY (87) **Directory not empty**

Cause: A directory with entries other than "." and ".." was supplied to a remove directory or rename call.

EDQUOT (88) **Disk quota exceeded**

Cause: One of the following errors occurred:

A write to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because the user's quota of disk blocks is exhausted.

The allocation of an inode for a newly created file failed because the user's quota of inodes is exhausted.

ELIBACC (89) **Shared library cannot be accessed**

Cause: A shared library referenced by an executable file cannot be found.

ELIBBAD (90) **Shared library format is bad**

Cause: A shared library referenced by an executable file cannot be linked because it is not in the correct format.

ELIBSCN (91) **.lib section in executable is corrupted**

Cause: The **.lib** section in an executable file cannot be processed.

ELIBMAX (92) **Too many shared libraries**

Cause: An executable file attempts to link to more than 10 shared libraries.

ELIBEXE (93) **Reserved**

ESITEDN1 (94) **Required site is not available**

Cause: A site requested or required for an operation is not available.

ESITEDN2 (95) **Operation terminated because of site failure**

Cause: During the processing of the system call a required site became unavailable. It may be that there is a network hardware failure and that processing on the remote site is continuing. The operation may or may not have completed.

ENOSTORE (96) File or working directory is unavailable

Cause: An attempt is made to access a replicated file, including a component of the path prefix or the current working directory, that is not available within the current partition. This can occur when the directory entry for the file is more widely replicated than the file itself. Under these conditions the directory entry can be found when the file is not present. This error is also returned when the current working directory becomes unavailable because of a site failure or unmount.

ENLDEV (97) Not a local device

Cause: A remote device is specified in a mount or umount system call or an attempt is made to open a remote character device that is not a TTY.

EBADST (98) Bad site specification

Cause: A system call required the specification of a site and the value provided is invalid or the site specified is not available.

ELDWRG (99) Load module not for this machine type

Cause: An attempt is made to execute a load module on a site that is a different machine type.

ELOCALONLY (100) Operation restricted to local site

Cause: A process is attempting to change execution sites while it is using facilities that are not network transparent.

ELOCK (101) Lock conflict

ETABLE (102) Miscellaneous system table full

Cause: The system has run out of space in some internal table.

EXGFS (103) Improper mount operation.

Cause: An attempt was made to mount a file system in a way that is inconsistent with previous mounts. This error could be caused by two file systems (which are not marked as replicated copies of one another) having the same global file system (gfs) number or by other inconsistencies such as replicated copies of a file system being mounted on different directories.

EPBUSY (106) Pack is busy

Cause: In the **mount** system call, a pack is specified that is already mounted.

ENSPEC (107) Not a specific site

Cause: A storage site operation was attempted at a site that is not the specific site required for the operation.

EDLOCK (108) Directory in conflicting mode

Cause: An operation was attempted on a directory that is locked in a manner that is not compatible with the desired operation.

ENOSYS (109) Function not implemented

Cause: An attempt was made to use a function that is not available in this implementation.

B.0 Appendix B. Writing a Queuing System Backend

This section provides information for writing a queuing system backend. It assumes you have a basic understanding of queue backends, friendly backends, and unfriendly backends. Only friendly backends are discussed; the term "backend" in the following discussion refers specifically to friendly backends, not backends in general. See *Managing the AIX Operating System* for more information about backends.

Subtopics
B.1 Introduction

B.1 Introduction

The principal purpose of a backend is to send characters to a device, typically a printer. There are several ways the backend can do this. First, it can open a particular device and write to it. This has the advantage of simplicity, but it means that the backend cannot be used for any other device. Second, it can accept a parameter supplied by the user to tell it which device to use. This is more flexible, but involves a little extra work. Third, it can simply write to its standard output, and the **qdaemon** command will automatically open the device onto the correct file descriptor. This is the recommended method. It works only if the **file** field in the **qconfig** file has been set up appropriately.

The backend is invoked once for every file or group of files to be printed. The name of each file to be printed is passed to the backend as a parameter. The backend must open the file, read its contents, and send them to the device in one of the ways previously described.

Since the backend must open files, read them, and write to devices, you (the writer of a backend) should understand the domain where the backend operates. When a backend is invoked, its current directory is the one where the print request was made. The name of the file or files to be printed can either be a direct or relative path name. The user ID and group ID of the backend are those of the process that invoked the **print** command.

If the backend writes to its standard output and allows the **qdaemon** process to open the device, permissions are handled by the **qdaemon** automatically. Otherwise, the backend will need to have write permission on the special file corresponding to the device. This may require changing the protections on the device or installing the backend set-user-ID or set-group-ID.

By default, **stdin, stdout,** and **stderr** are all open to the null device (**/dev/null**), though it is possible to override the setting of **stdout** (and possibly **stdin**) with the **file** and **access** lines in the **qconfig** file.

Subtopics

B.1.1 Interaction Between Qdaemon and Backend

Besides reading files and writing to devices, a friendly backend must cooperate with the **qdaemon** in several ways. The requirements can be summarized as follows:

Recognize a **-statusfile** parameter and call a library routine that does some initialization.

Print burst pages as requested

Print extra copies as requested

Update status information (pages printed, percentage done periodically.

Supply charges (accounting data) for the completed job

Exit with some agreed on codes

Pass error messages through a special routine

Set state to WAITING, if appropriate

Terminate cleanly on receipt of SIGTERM

Each requirement is discussed more fully in the following.

There is a set of library routines that the backend should use to fulfill these requirements. The routines were designed to make the task of writing a backend as easy as possible. These routines are in the **/usr/lib/libqb.a** library and accessible with the **-lqb** flag. The individual routines are discussed in the body of the text that follows, and a summary table is given at the end of this chapter.

B.1.2 The -statusfile Parameter

When the **qdaemon** process invokes a backend, it passes the following parameters, in order:

1. The parameters appearing in the **qconfig** file
2. The **-statusfile** parameter, if running as a friendly backend
3. The flags that the **print** command did not recognize, in the order they were given
4. The names of one or more files to be printed.

The presence of the **-statusfile** parameter indicates that the status file is open on file descriptor 3 of the backend.

The status file provides a means for the **qdaemon** process and the backend to communicate. The daemon passes such information as the date of the file, which burst pages are to be printed, the number of copies to be printed, and so on. The backend passes back the charge for the job it has just finished running. In addition, the backend periodically writes into the file the number of pages it has printed and what percent of the job is finished. This information is read by the **print -q** command.

Backends should never explicitly write into their status file. Instead, they should call the library routines that do so. The reason for calling the routines is twofold: (1) backends are spared the trouble of accessing the status file directly, and (2) the format of the status file can be changed without requiring backends to be rewritten. In this case, the backends only need to be re-linked.

To initialize certain data common to the library routines, the backend must call the routine **log_init**. The call is:

```
log_init();
```

This routine should be called when the **-statusfile** parameter is recognized. The **log_init** routine, like all the routines in library whose names begin **log_**, returns a value of -1 if it fails.

B.1.3 Burst Pages

There are four types of burst pages:

header A page preceding a file that shows its title, date, recipient, and other information.

trailer A page following a file that gives the name of the user of the output.

feed Blank pages printed only when the printer has become idle. Feed pages make it easier for users to tear off paper from the printer.

align A form-feed printed only when the printer has been idle and is about to print a new job. The form-feed aligns the paper to top-of-form and is helpful if someone has moved the paper while the printer was idle.

If the backend will never print any burst pages, the following information can be skipped.

The printing of burst pages is done automatically by the **burst_page** routine. The routine takes two parameters: the address of a function and the width of the header and/or trailer desired. The function is called to put each character of the burst pages to the device. The parameter to this function is the single character to send. If the function address is NULL (**#include <stdio.h>**), the routine uses the supplied function.

By passing the address of a special function for output, a backend can maintain strict control of what goes to the device and when it goes to the device. For example, the **burst_pages** routine uses line-feeds to separate lines, and form-feeds to separate pages. If the device requires a carriage return to precede every line-feed, the special function can make such a translation.

The basic algorithm for synchronizing calls to the **burst_page** routine with file printing looks like this:

```
(1)  burst_page(fnaddr, width);
     while (files are to be printed)
     {
(2)  burst_page(fnaddr, width);
      print the next file;:
(3)  burst_page(fnaddr, width);
    }
```

Every backend should follow this structure. The line numbers are used for reference in the following explanation.

The **burst_page** routine uses the information in the status file to decide whether (and how) to print a header, a trailer, some feed pages, or an aligning form-feed. The status file is set up by the **qdaemon**, using the information provide in the **qconfig** file. For example, if the **qconfig** file contains the line **header=group**, the call on line (2) results in a header page only if this file is used by a different user than the user who printed the previous file on this device. The **burst_page** routine when invoked on line (2) makes that test and either prints the header or returns. Similarly, line (3) either prints a trailer or does nothing.

With the exception of line (1), which may appear to be extraneous, the algorithm is simple. This first call is necessary because **qdaemon** does not ask the backend to print a **group** trailer until it knows positively that there are no more files for a particular user. It cannot know this fact until either the first file for the next user is ready to be printed or there are no more files for this device. In the first case, **qdaemon** appends the trailer request for the previous user to the file request for the current one. Line (1) prints the trailer for the previous user if the **trailer=group** option has been selected; otherwise, it does nothing. In the second case, the backend is invoked with no file parameter at all. In this case, line (1) prints both a trailer and feed pages (assuming **qconfig** requests them), the **while** test fails, and the backend exits.

The **burst_page** routine assumes that the printer is at the top of the page, and it prints a form-feed at the end of its header or trailer to leave the printer in the same state. Backends are responsible for maintaining the position of the paper. The **align** option is useful only for device like continuous-form daisy-wheel printers, where it is possible for the printer paper to be out of alignment after a job is removed.

The **burst_page** routine should be enough for most friendly backends. If it is not, the library provides a set of routines at a lower level that should prove helpful for generating burst pages. There is a group of routines that return information from the status file, and two other routines that print headers and trailers, respectively.

Functions in the first group take no parameters; the following describes their actions:

get_align Returns TRUE or FALSE, telling whether an alignment form-feed is to be printed, assuming **get_newuser()** is TRUE and **get_endgroup()** is FALSE.

get_endgroup
Returns TRUE or FALSE, telling whether this is the end of a group of files for the same user.

get_feed Returns the number of feed pages to be printed, assuming **get_endgroup()** is TRUE and **get_newuser()** is FALSE.

get_from Returns the name of the person that made the print request.

get_header
Returns NEVER, ALWAYS, or GROUP (**#include <IN/backend.h>**), depending on the configuration.

get_lastuser
Returns the name of the previous user, assuming **get_endgroup()** is TRUE.

get_moddate
Returns a string showing the modification date of the file.

get_newuser
Returns TRUE or FALSE, telling whether this is the beginning of a group of files for a new user.

get_nodeid
Returns the node ID.

get_qdate Returns a string showing the date that the request was queued.

get_title Returns the title of the job being printed.

get_to Returns the name of the person for whom the job is intended.

get_trailer
Returns NEVER, ALWAYS, or GROUP, depending on the configuration.

In addition, there is a routine **put_header(fnaddr, width**), that prints a header with no following form-feed, returning the number of lines printed, and a routine, **put_trailer(user,fnaddr,width**), that prints a trailer for **user**, again with no following form-feed, and returns the number of lines printed. The **fnaddr** and **width** parameters work like the same parameters in the **burst_page** function previously stated.

It should be emphasized that the auxiliary functions should not be necessary for most backends. The **burst_page()** routine handles all tasks required when it is called as described in the previous algorithm.

B.1.4 Extra Copies

The user can request that extra copies of a file be printed with the **print -nc** command. The **print -nc = 5 filename** command prints 5 copies of a file.

The **print** program passes the **-nc** information to the **qdaemon** process, which puts it into the status file. Backends should get the information by calling the **get_copies()** routine, which returns the total number of copies desired.

B.1.5 Job Status Information

The **print -q** command displays information about each currently running job, including its originator, its title, the number of pages to be printed, and the percentage completed. All this information comes from the status file. Most of the information is set up by the **qdaemon** process when the backend is first invoked, except the **pages printed** and **percent done** fields, which must be filled in by the backend itself.

To provide this information, the backend should periodically call **log_progress(pages, percent)**, which writes the two numbers in the appropriate place in the file. The backend is free to call this routine as frequently or infrequently as desired; once at the end of each page is recommended.

B.1.6 Charge for the Job

Whenever a backend completes a job, the **qdaemon** process reads the status file for a charge. If the **qconfig** file has been set up appropriately, the charge is written to a file that is eventually processed by the accounting programs, resulting in a bill (real or imaginary) for the user issuing the print request.

The backend passes the charge back to the **qdaemon** process with the routine **log_charge(charge)**, where **charge** is a long integer. The backend should certainly call this routine on exit. It should also call the routine along with **log_progress** while printing the job. Otherwise, if the job is canceled, no charge will be made for the pages printed up to that point.

The charge is interpreted by all current accounting programs as the number of pages printed. However, a backend might decide that one page on its device is worth two or three normal pages (or some fraction) and set the charge accordingly.

B.1.7 Exit Codes

When a backend exists, the **qdaemon** process looks at its exit code for information about whether the job was completed successfully, whether the device is still usable, and so on. Therefore, it is important that backends use the same convention for their exit codes. The backend should use **#include <IN/standard.h>** for the values of the codes mentioned here.

The permissible exit codes are:

EXITOK No problems were encountered.

EXITBAD The parameters were bad in some way. That is, a flag was unrecognizable or illegal, a file could not be opened, and so on. The **qdaemon** process notifies the user, throws out the job request, and continues sending jobs to the device.

EXITERROR The backend could not finish printing the job and that it wants another chance. The **qdaemon** process restarts the same job (from the beginning) on the same device. The **qdaemon** process enforces a limit on the number of times that the job will be restarted.

EXITFATAL The job could not be finished because of a problem in the device that requires manual intervention. The **qdaemon** process sets the state of the device (displayed by **print -q**) to OFF, sends a message to the console, and does not run any further jobs on that device until someone has explicitly set its state to ON again (with a **print -du**).

EXITSIGNAL
The backend was interrupted by a **SIGTERM** signal (**#include <signal.h>**).

B.1.8 Return Error Messages

If the backend cannot run a job (that is if it exits EXITBAD or EXITFATAL), it should send a message to the **qdaemon** process explaining the problem. The **qdaemon** process passes the message to the user, and, for EXITFATAL prints it on the console.

The message should be sent with the **log_message** routine, which takes parameters in the style of **printf**:

```
log_message("cannot open file %s; error return %d\n",
                filename, erret);
```

The message cannot be longer than MAXMESG (**#include <IN/backend.h>**) bytes.

B.1.9 Set State to WAITING

The **print -q** command displays the status of a particular device. One of the entries in the table that is displayed shows whether the device is READY, RUNNING, WAITING, or OFF. This information is taken from the status file.

Normally, the **qdaemon** process keeps the status file updated, and a backend need never worry about it. However, some backends may want to explicitly set the state to WAITING (**#include <IN/backend.h>**) if they can no longer send output to the device, and set it back to RUNNING when output resumes. For example, a backend that paused at the end of each page, waiting for the user to load the next page and type a RETURN, might want to set the status to WAITING during this time.

The **log_status(status**) routine can be used to change the status of the job from RUNNING to WAITING and back again. The parameter is the new status.

B.1.10 Terminate on Receipt of SIGTERM

When a user cancels a running job with **print -ca**, the **print** command passes the request to the **qdaemon** process. Therefore, in order for cancellation to work, the backend must terminate soon after receipt of the signal. There are two ways to comply with this requirement.

First, the backend can do nothing special about SIGTERM, in which case the signal kills the backend process immediately. This option is the simplest, but does not allow the backend to do any cleanup (reset line speeds, put paper at top-of-form, hang up the phone) before it terminates.

Second, the backend can catch **SIGTERM**, carry out whatever cleanup tasks are required, and exit **EXITSIGNAL** (**#include <IN/standard.h>**). The special exit code tells the **qdaemon** process that the job was canceled.

Backends that decide to catch **SIGTERM** should exit very soon after receipt of the signal. If the cleanup code is too long, or if it can wait indefinitely (for terminal to open, for a device to respond, and so on), the backend is not friendly.

B.1.11 Backend Routines in libqb

The following is a list of backend routines available using the **ld** or **cc** command-line option **-lqb**.

```
burst_page(fnaddr,width)
  int (*fnaddr)();

get_align()

get_copies()

get_endgroup()

get_feed()

char *
get_from()

get_header()

char *
get_lastuser()

char *
get_moddate()

get_newuser()

char *
get_nodeid()

char *
get_qdate()

char *
get_title()

char *
get_to()

get_trailer


log_charge(charge)
  long charge;

log_init()

log_message(...)

log_progress(pages,percent)

log_status(status)

put_header(fnaddr, width)
  int(*fnaddr)();

put_trailer(user, fnaddr, width)
  char *user;
```

```
int (*fnaddr)();
```

C.0 Appendix C. Writing Device Drivers

Subtopics

C.1 Introduction

Device drivers are programs that exist inside the AIX kernel. They connect physical devices, such as printers, displays and disk drives, to an AIX application and control an application's use of those devices. AIX applications send and receive data from device drivers via system calls and sockets. Although the interface to the physical device can be complicated, your interface to the device driver is simple and flexible, and can insulate your application from hardware dependencies.

When an application performs an I/O to a physical device, the following sequence of actions occurs:

1. The application opens a special file associated with the physical device.

2. The device driver prepares the device for the I/O. The application may optionally configure the device driver and device through **ioctl** routines

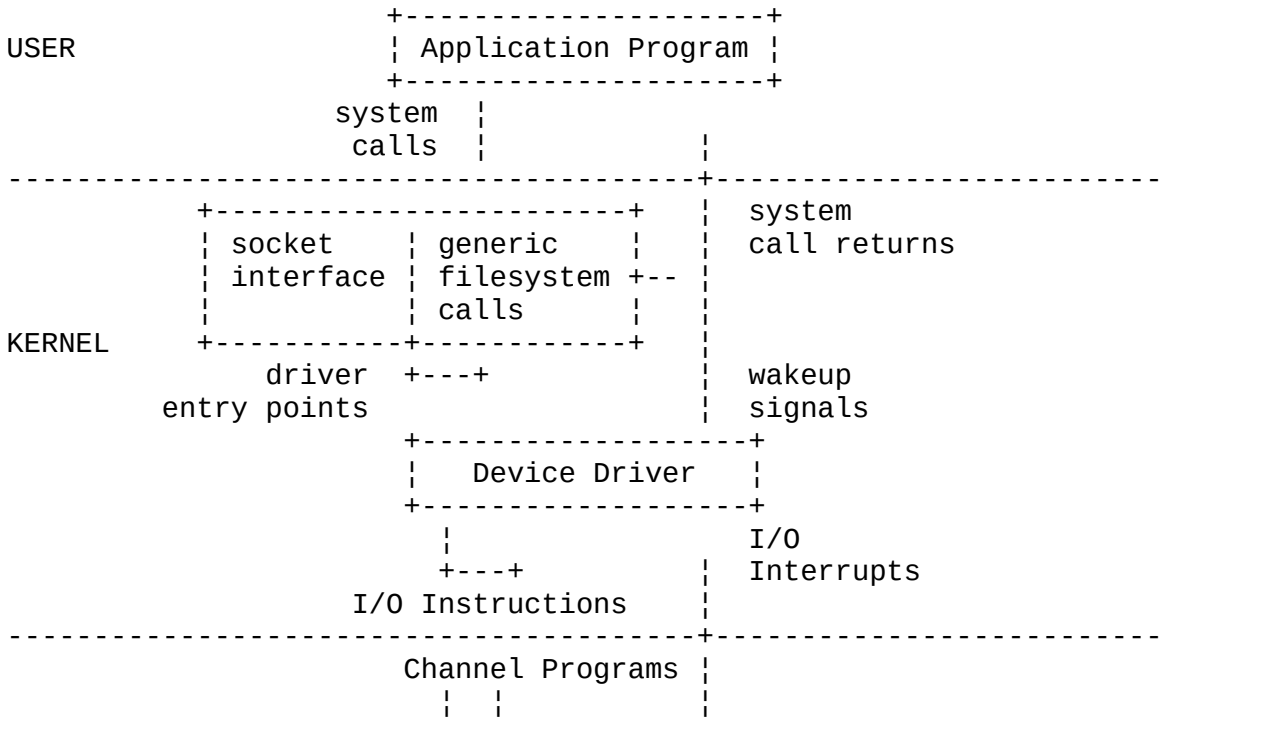
3. The application reads and writes data to or from the device driver

4. For AIX/370, the device driver sends or receives the data to or from the hardware by initiating channel programs.

5. For each I/O operation that the device driver performs, the device driver usually receives one or more interrupts in return. This is device dependent.

6. When the application finishes performing the I/O, it closes the special file so that other applications may use the same device if it cannot be shared.

Figure C-1 and Figure C-2 illustrate this process for the AIX/370 and AIX PS/2 systems.



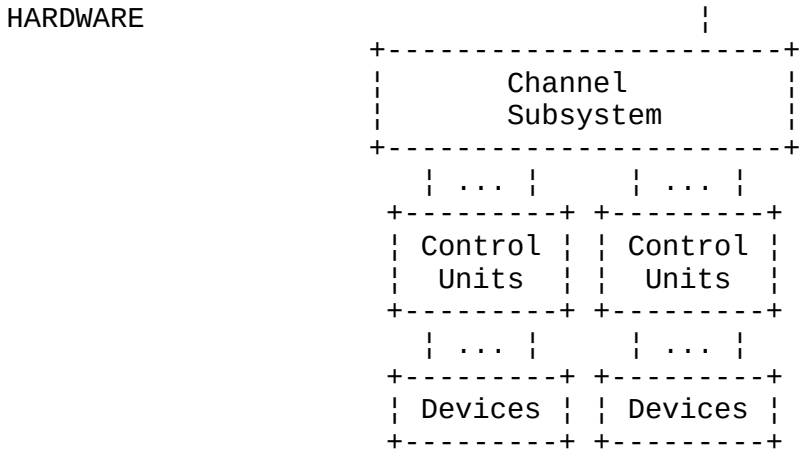


Figure C-1. AIX/370 Device Driver Model

The AIX kernel provides a general framework for writing device drivers. This appendix describes that framework in terms of device driver concepts, device driver types and their data structures. It also provides information on kernel subroutines, macros, and the kernel debugger as well as material on driver configuration and initialization.

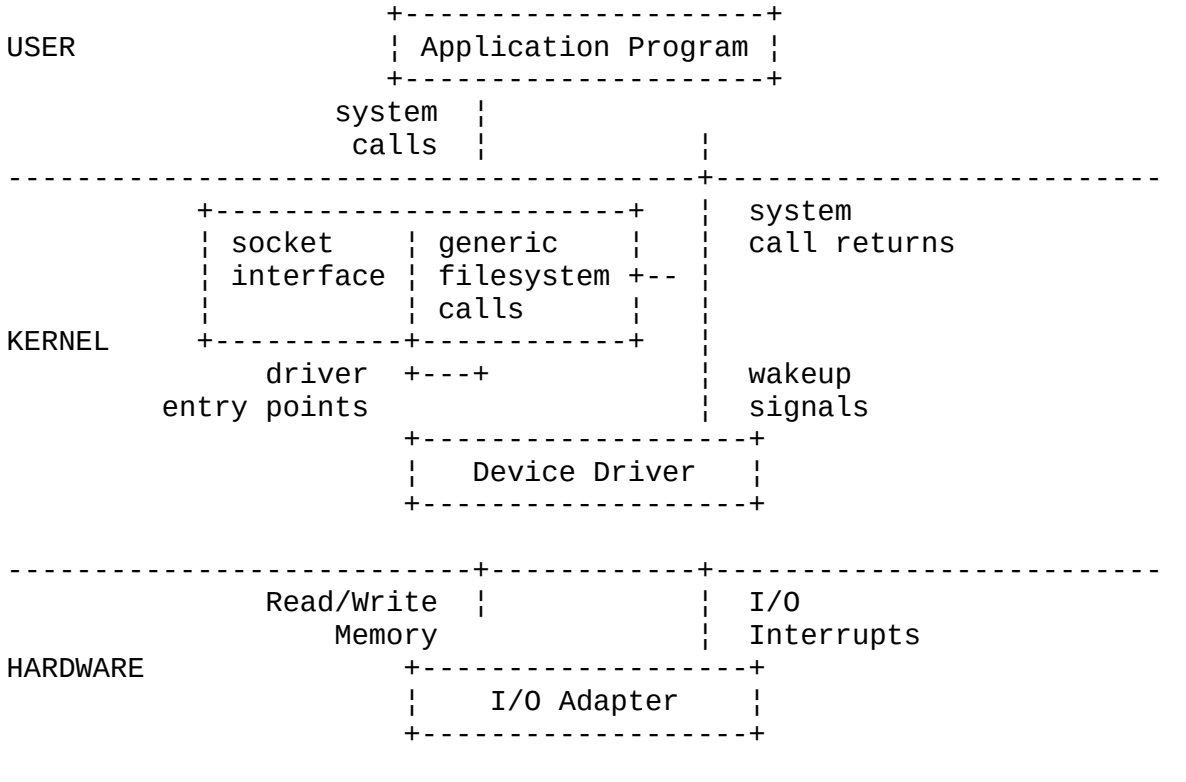


Figure C-2. AIX PS/2 Device Driver Model

C.2 Device Driver Concepts

This section presents a few basic concepts that underlie the AIX framework for device drivers. The following driver concepts are presented:

General considerations in AIX device driver

Entry point

Major/minor device numbers and special file

Multiplexed device

Autoconfigured and non-autoconfigured device driver

Header files used in device drivers

Subtopics

C.2.1 General Considerations in AIX Device Drivers

Some general considerations in AIX devices drivers involve non-preemption, context, buffering data, transferring data to a device, deadlocks, and races.

Subtopics

C.2.1.1 Non-preemption

A process executing in the kernel cannot be pre-empted by another process. A process in a device driver is pre-empted when one of two events occurs:

A user-page fault occurs - the process attempts to access a page o memory that is currently swapped out of memory

The process puts itself to sleep

A process can be suspended, but not pre-empted, at any time by interrupts above the level that has been masked.

Interrupt handlers can set the **runrun** external variable to a nonzero value to cause the scheduler to dispatch the next procedure with the highest priority. This technique is used when you think that some process, other than the caller, is more deserving of a time slice.

C.2.1.2 Context

There are two contexts of an AIX device driver:

Synchronous process	The device driver can reference user address space and process-specific parameters, such as those stored in the **u** and **proc** structures. In process context, the device driver may temporarily suspend execution on an event. An interrupt can occur while any process is running.
Asynchronous interrupt	Also referred to as real-time context. In interrupt context, the device driver cannot reference user address space or process-specific parameters. During real time, the device driver can keep track of which process or user owns the request by referring to static or global data areas. During interrupt context, the driver cannot sleep (be suspended) since only a process can sleep.

Design your device driver to do the bulk of its work in process and not interrupt context. During real time the device driver must be very efficient.

Consider carefully what type of interrupts need to be disabled and for how long. Interrupts should be masked over small intervals where code re-entrance can occur and global data structures are updated.

Disabling interrupts over extended periods of time is not advisable. Loss of data, in other device drivers as well as yours, can occur if you mask interrupts for too long or the code path of your interrupt handler is excessive.

C.2.1.3 Buffering Data

The two places where data is stored for a device driver are ***user space*** and ***kernel space***. User space is the data area of AIX processes. Device drivers use user space to store large chunks of data for direct memory access (DMA) and to store data that is not needed during interrupt handling.

Manipulate user data only in the context of the requesting process. Do not use user space in interrupt context to store real time transactions because AIX does not guarantee that the device driver will be notified when the process dies. More importantly, the user space will not be there when the process is not running.

Kernel space is generally reserved for data that is manipulated during interrupt context or outside of the calling process. Kernel space is also used to store data that is frequently used by the device driver.

Note: Never use dynamically allocated storage (such as variables allocated within a procedure and stored on the stack) to buffer transfers that can be pre-empted. The virtual address space allocated to the stack is shared by the currently running process.

C.2.1.4 Transferring Data to a Device

Transferring data to a device can take three forms:

Large DMA transfers to and from kernel address spac
Small transfer
Large DMA transfers to and from user space

When performing large transfers of data in kernel address space, the device driver sets up the DMA controller and then initiates the transfer. Assuming that the device will generate an interrupt at the end of the transfer, the interrupt handler then checks for and handles any errors from the previous I/O, initiates retry or posts completion, pulls the next request from the queue and initiates the required I/O operations.

Small data transfers should be rebuffered through kernel space in process context. Once copied into kernel space, the device driver can then handle the I/O in interrupt context.

During large DMA transfers from user space, synchronous code must pin (prevent the physical pages form being disassociated from the requesting process's virtual address space) the buffers in memory. Once pinned, the buffers can be treated as kernel space, provided the process remains in the device driver during the transfer.

There are some deadlock and performance risks while using DMA from user space. These are:

Locking down too much memory slows system performance

Too many lockdowns can cause memory deadlock

For these reasons, DMA requests from user space should be broken down into smaller chunks which should be no larger than the size of a page (4096 bytes). In addition, you should limit the number of queued DMA requests in your device driver and the total amount of pinned memory.

C.2.1.5 Deadlocks and Races

A ***deadlock*** is a situation in which two processes are competing for the same two resources. For example, if process A has resource X and needs resource Y while process B has resource Y and needs resource X, then a deadlock occurs. Resources for which processes compete include:

System buffer
Inode
Memory pages and swap spac
clist
mbuf
Net-messages

Deadlocks can be avoided in the following ways:

Tentative allocations — The requesting process performs a non-blocking allocation on resource X and, if successful, performs a non-blocking allocation on resource Y. If the process does not get resource Y, it gives resource X back and tries again later. A deadlock is avoided because a process does not stubbornly hold onto a resource while trying to get another.

Advance reservations — A process performs advanced reservations on resource A and then resource B. If the process gets both A and B, it locks them. If the process does not get either, it cancels its reservation. If a process can do a non-blocking lock (make a reservation), a deadlock can be avoided.

Request ordering — Only applicable to distinguishable resources like ports on a communications adapter. In such cases, the device driver must define the order in which the resources must be allocated and always adhere to that order.

A ***race*** is an event involving two processes that may be executing the same or related algorithms, and the outcome of the computation is different depending on the process and when it executes. There are three types of races:

Fair — If the code is properly serialized, the race will not compromise the correct execution of the algorithms. The race occurs between two processes and the code works regardless of the process.

Wasteful — System resources are wasted when many processes are summoned and forced to compete for such things as a free buffer in the buffer cache. If a long line is unlikely, summoning every process that is waiting for the resource might be easy to do with a low expected cost. If, on the other hand, delays are expected, you should devise some auxiliary serialization mechanism to avoid scheduling multiple processes when only one process can continue execution. One possible mechanism is to use the **wakeup_one()** subroutine which schedules only a single process. The **select** system call has this feature: when only one process is first in line and when the requested I/O becomes available, only that one process is executed,

no matter how many other processes are in line.

Unintentional

Unintentional races (bug race conditions) are bugs in parallel algorithms that result in incorrect execution depending on the sequencing of the parallel executions. The classic unintentional race is as follows:

Process 1	**Process 2**
1a) Obtain resource, if available	2a) As a result of interrupt, mark resource as available
1b) Set resource WANTED flag	2b) If resource WANTED has been marked, reschedule the requesting process
1c) Process relinquishes CPU while waiting for resource	

If steps 2a and 2b happen between steps 1a and 1c, Process 1 will never be rescheduled for execution. The code contains an unintentional race condition. The easiest fix is to disable interrupts between steps 1a and 1c.

C.2.2 Entry Points

When an AIX application issues an I/O or a socket system call, the AIX kernel calls a device driver to perform the requested operation. Each device driver is invoked by the AIX kernel using standard entry points: **ddinit**, **ddreset**, **ddopen**, **ddclose**, **ddioctl**, **ddread**, **ddwrite**, **ddstrategy**, **dddump** and **ddselect** where **dd** is a prefix that uniquely identifies a device driver.

Your program does not directly enter the device driver's entry points. There is kernel code between the system call and the device driver that may perform one or more of the following operations:

Buffer dat

Interpret network protoco

Hold or maintain buffers of character

Ensure that the system call parameters are valid

A device driver is actually linked into the AIX kernel load module and is therefore part of the kernel. The procedure for rebuilding the kernel involves constructing a table that contains pointers to the entry points for each of the device drivers in the system. This table is called the device switch table, and it is an array of type **struct devsw**, which is defined in the **/usr/include/sys/conf.h** header file. The device switch table is constructed using information from the **/etc/master** and **/etc/system** files. A stanza in the **/etc/master** file for each device specifies the unique **dd** prefix and lists the entry points that are defined for the device.

A device driver can also include a routine to service interrupts that an I/O device generates when an operation is finished. Such an entry point is called the interrupt handler. It is named **ddintr** by convention. The **ddintr** entry point is not included in the device switch table; instead, it is identified to the kernel with the **intrattach** kernel subroutine, which is usually called from within the **ddinit** entry point.

A device driver does not have to have all of the previously-mentioned entry points. If an entry point is not included in a device driver, then either the **nulldev** or **nodev** subroutine is substituted into the appropriate position in the switch table.

C.2.3 Major/Minor Device Numbers and Special Files

Each driver is identified by an unique index in the device switch table called a major number. A minor number selects a particular sub-unit or options in a given device driver.

An application can access a device's major/minor number combination through a special file. Special files are inodes with major and minor numbers. A special file is categorized as either block or character by the **mknod** command.

Access to a device driver is controlled by the permissions assigned to its special file. For example, if a special file's permissions are **-rw-r--r--** and it is owned by user root, only root applications have write permissions to the driver while every application can read.

A special file may be multiplexed, where the minor number is further broken down into channels. A channel may designate a smaller sub-unit or option than a minor number. For more information on major and minor numbers, see "Requests for Device I/O" in topic 1.1.6.10.

C.2.4 Multiplexed Devices

A multiplexed device is one whose device driver allows the path name of its special file to be followed by a character string that specifies additional information. This path name extension is frequently used to identify a logical or virtual sub-device, called a channel, or to select device options. Only character devices can be multiplexed.

For example, the special file for the console virtual terminal device driver is named **/dev/hft**. Each time a new virtual terminal is opened, it is assigned the name **/dev/hft/n**, where **n** is an ID number assigned to that virtual terminal. Multiplexing provides a means of accessing individual driver sub-units, without having to create additional special files in **/dev**. In addition, multiplexing allows the device driver to allocate tables and arrays on a per-need basis. For more information on implementing multiplexed device drivers, see page C.4.1.1.

Multiplexed channel numbers are available to all character device entry points except the interrupt handler **ddintr**.

C.2.5 Autoconfigured and Non-autoconfigured Device Drivers

With very few exceptions, all device drivers in AIX are autoconfigured. Autoconfigured device drivers are capable of determining which hardware units are present and the associated addresses of those units.

A device driver is autoconfigured for three reasons:

All device drivers autoconfigure based on the programmable optio select (POS) registers located on each adapter. Via POS, the driver can determine:

- Available hardware units
- Port interrupt levels and direct memory access (DMA) arbitration levels
- Other configured options, such as on-board memory address, amount of memory, port addresses, and so forth.

Thus, autoconfigured device drivers can determine hardware characteristics independent of configuration programs like **config**, **devices**, and **minidisks**. This simplifies system configuration above the AIX kernel.

The size of the AIX kernel is limited by physical memory Autoconfigured device drivers reduce the amount of statically allocated data structures in the kernel because these device drivers can allocate their tables and data structures based on the number of hardware units available.

Dynamic allocation is preferable to static allocation because it suits actual needs, rather than worse case. Of course, the number of tables can be tuned without regenerating the system. The allocation of large tables should be deferred until open time.

A C compiler is not supplied with the base AIX Operating System Therefore, **conf.c**, the file containing kernel configuration information such as the device switch table and variables containing static table sizes, could not be recompiled.

A device driver that can not be completely autoconfigured is referred as a non-autoconfigured device driver. This class of device drivers includes drivers that:

Cannot determine the number of units attache
Must allocate static storage for the maximum possible number of unit
Cannot determine the existence of their devices at auto-configure tim
Must always be present in the system

Non-autoconfigured device drivers statically allocate their data structures and require users to provide their configuration information. Non-autoconfigured device drivers also always have their entry points in the device switch table.

C.2.6 Header Files Used in AIX Device Drivers

The following header files, which are located in the **/usr/include** directory, contain definitions of structures, constants, and data types that are used in device drivers. The header files common to both AIX PS/2 and AIX/370 systems are indicated as such.

Common File	Pertinent Contents
sys/param.h	Fundamental implementation parameters of AIX
sys/types.h	The **typedef**s defining data types used in kernel programming
sys/user.h	The definition of the user block structure
sys/tty.h	The structure definitions for character lists
sys/devinfo.h	The structure returned by the **IOCINFO** operation of the **ddioctl** entry point
sys/errno.h	Standard error codes used by AIX system calls
sys/conf.h	Device switch table structure definitions and device flag definitions used by autoconfigured device drivers
sys/proc.h	The definition of the **process** structure
sys/buf.h	Header for buffers in the buffer pool and otherwise used to describe block I/O buffers
sys/file.h	Definitions for flags passed to the **ddopen** entry point and definition for the file structure
sys/dir.h	The **dirent** structure which is used for looking at directories
sys/iobuf.h	The structure for the I/O buffer used by block device drivers
sys/ioctl.h	Structure definitions for **ioctl** arguments
sys/ioctlcmd.h	Macros for defining well-behaved **ioctl** routines
sys/systm.h	Important variable definitions contained in the AIX kernel
sys/machinfo.h	Structures used to determine machine type and configuration
sys/erec.h	Error record structure and definitions
sys/callout.h	The definition of the **callout** structure
sys/if_ieee802.h	Definitions related to sockets: types, address families, and options
sys/mdisk.h	Contains definitions for VTOC and provides minidisk support.

sys/netisr.h	Definitions for the **schednetisr** kernel subroutine
sys/if.h	Interface structure for network device drivers
sys/mbuf.h	The **mbuf** structure and macros
sys/socket.h	Definitions for socket address structure, protocol families
sys/space.h	Included by the **conf.c** file to allocate storage and initialize variables.

AIX PS/2 File	**Pertinent Contents**
sys/i386/pos.h	The **devdata** structure, contains power-on setup (POS) information for each adapter. It also declares definitions useful for interpreting the data in non-volatile memory
sys/i386/cmos.h	Contains definitions for accessing and interpreting the data in non-volatile memory
sys/i386/dmaralloc.h	Structures and definitions used to allocate DMA resources
sys/386/intr86.h	The **intrattach** routine request level masks
sys/i386/mmu386.h	Memory mapping macros
sys/i386/sufcfg386.h	Included by the **conf.c** file to define the number of pseudo terminals and machine-dependent routines.

AIX/370 File	**Pertinent Contents**
sys/b370/ccw.h	Contains the definitions for the command control words used to form channel programs.
sys/b370/csw.h	Contains the definitions of the fields composing the channel status word.
sys/b370/iohdlrs.h	
sys/b370/schib.h	Contains the definitions for the structure of the channel identification block
sys/b370/obr.h	Contains definitions for the Error Reporting
sys/b370/inccfgb370.h	Driver configuration headers
sys/b370/mdr.h	Definitions for EREP header
sys/b370/dump.h	Contains definitions of structures for support of System/370 kernel core dumps.

C.3 AIX/370 I/O Concepts

The System/370 I/O system was designed to off-load the CPU from I/O processing.

The I/O subsystem consists of the following components:

Channel

Subchannel

Channel pat

Control unit

Devices

A channel provides a means for connecting I/O devices to the CPU and memory. The channel executes a channel program and drives the control unit and devices. The channel manages data transfer, interrupts and control operations for the device.

A subchannel is the combined channel facilities for sustaining a single I/O operation. These facilities consist of internal storage for maintaining addresses, count, state, status and control information associated with the I/O operation.

A channel path is the physical data path capable of supporting subchannel to control-unit communication.

A channel containing multiple subchannels is capable of performing some degree of multiplexing.

A control unit provides the logic to operate and control an I/O device and to interface with a generic channel. The control unit decodes the commands received from the channel, interprets them and provides the signal sequences required for execution of the operation. It may provide any necessary buffering.

A device is any peripheral typically capable of interfacing with other machines, storage mediums, and input/output mediums. Printers, disks, tapes, and terminals are examples of peripherals. A programmer usually need not be aware of the control-unit/device boundaries.

The CPU accomplishes I/O by starting channel programs on specific channels connected to the requested device. Each subchannel is associated with a single channel path.

A channel program consists of one or more command control words (CCWs) outlining the operation to be performed by the device. Each CCW contains the following fields:

Command-code
Data address
Flags
Count.

The *command-code* field contains generic channel operations combined with device-specific commands. Listed below are examples of generic channel

commands:

read	Causes data to be transferred from the device to main memory.
read-backwards	May only have meaning for specific devices (such as tape).
write	Involves a transfer of data from main memory to the device.
sense	Transfers up to 24 bytes of device information from the device to the channel.
control	Does not involve any data transfer other than the command itself.
transfer-in-channel	Causes a branch in execution within a channel program.

Certain operations (search commands) cause the channel to fetch the **n+1th** command rather than the next sequential command. By using the **transfer-in-channel** command, looping and branching are implemented. For example, the following channel program segment would cause the **search** operation to occur successively until the key is matched:

```
Search Key Equal
Transfer-in-channel (to previous command)
Read Data
```

The **data-address** field specifies the destination or source in memory, or an indirect address word (IDAW) used for specifying virtual addressing.

The **flags** field contains numerous options such as:

```
Data chainin
Command chainin
Suppress incorrect length indicatio
Skip (don't perform data transfer
Program-controlled-interrup
Indirect-data-addressing (use of virtual memory)
```

The most common are command chaining and data chaining. Command chaining allows the channel program to consist of multiple CCWs, by chaining them together. Data chaining allows a channel operation to be described by multiple data vectors (scatter-gather). The **count** field is the number of bytes involved with the transfer (32k maximum).

I/O operations are initiated using the I/O instruction set and various dedicated low-memory locations. For example, the **Start-I/O** operation requires that the channel address word (CAW) gets filled with the address of the channel program; the operand to the instruction is a word containing the number of the channel, subchannel and I/O device.

All I/O operations are essentially asynchronous; once the CPU initiates a channel program, the channel begins execution of the channel program without CPU intervention. However, certain commands reach completion as soon as they are initiated and could be considered synchronous.

The completion of a successful I/O operation is signified by the channel generating one or more I/O interrupts and posting its completion status in

a set of memory locations in low memory called the channel status word (CSW). Alternatively, the CPU can poll the channel by executing an instruction to test the status of the channel which also causes a CSW to be stored (for example, when interrupts are disabled).

A channel may cause an interrupt after any of the following conditions:

Channel end — Indicates that the device has completed its use of channel resources as is known, as the primary-interruption status.

Device end — Indicates that all device activity is finished.

Control unit end — Indicates that the control unit has completed its interaction as is known, as the secondary-interruption status.

Attention

A device causing interrupt may contain combinations of the above conditions. Interpretation of the conditions is somewhat device dependent.

Typically, all indications are simultaneous with the exception of devices such as a tape unit, which may free the channel long before the device finishes (as in rewinding the tape).

There is a single CSW used by all channels to report I/O completion. The interrupt handler for an I/O interrupt is responsible for saving the CSW before re-enabling interrupts.

The CSW contains the following fields:

CCW address — Last CCW successfully executed + 8

Unit status — Device status

Channel status — Channel status

Count — Residual transfer count

Deferred condition code

The CPU directs channel activity using a set of I/O instructions. These I/O instructions provide the ability to:

Initiate an I/O operatio

Halt an I/O operation or devic

Clear the channel or I/O operatio

Test the status of a channel or I/O operatio

Query channel identification

Each I/O instruction sets a command completion code in the processor status word (PSW). The completion code (or CC) represents the success of the I/O instruction. There are four CCs roughly equivalent to:

```
0         Operation started or completed successfully
1         CSW stored
2         Busy (operation not started)
3         Not operational.
```

The meaning of each condition code for each instruction is documented and is instruction-specific.

Subtopics
C.3.1 370-XA I/O
C.3.2 AIX/370 Device Drivers

C.3.1 370-XA I/O

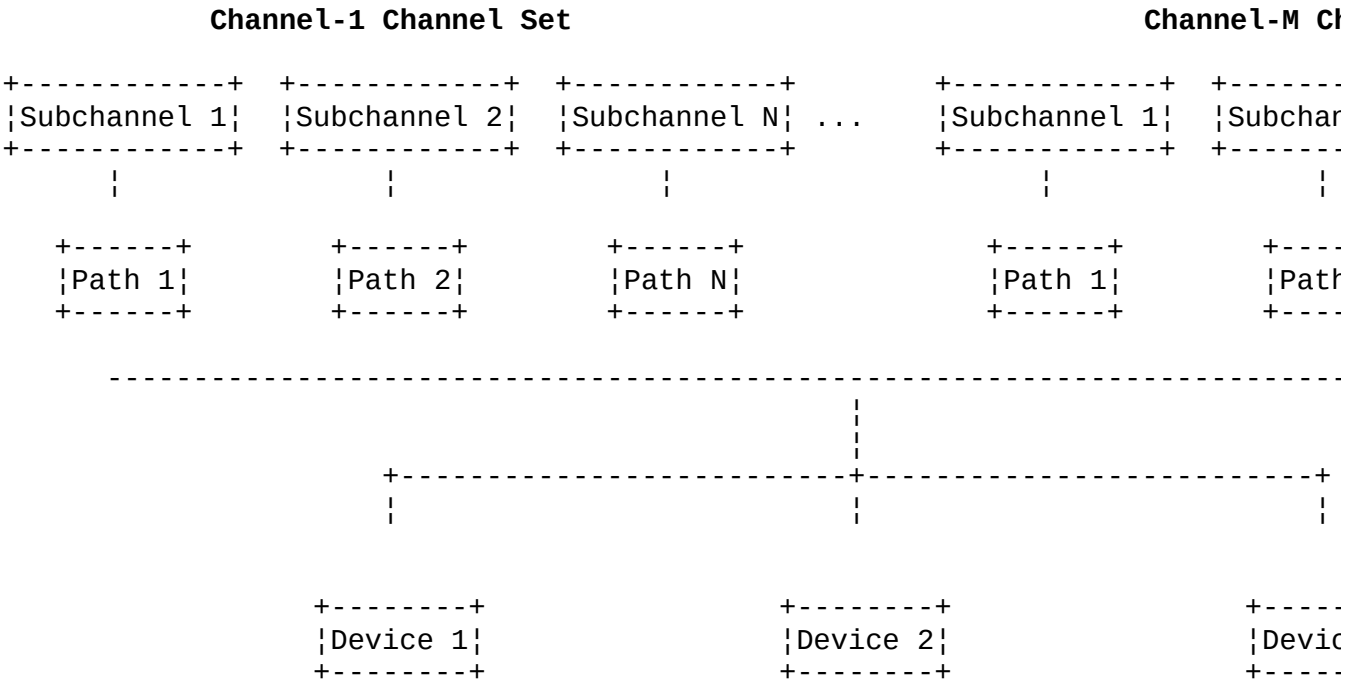
In the extended-architecture system, a number of major enhancements were made to the I/O facilities. These include:

- Path-independent addressin
- Path managemen
- Dynamic path reconnectio
- Interrupt subclasse
- 31-bit addresse
- Address-limit checkin
- New more powerful I/O instruction
- Suspend/Resume capabilit
- Monitoring

Standard AIX/370 channels programs run unmodified on the 370-XA system; however, AIX/370 I/O instructions are not supported.

In the AIX/370 system, each subchannel is associated with only one channel path, whereas in the 370-XA mode, each subchannel is uniquely associated with a single I/O device. This is illustrated in Figure C-3 and Figure C-4.

AIX/370

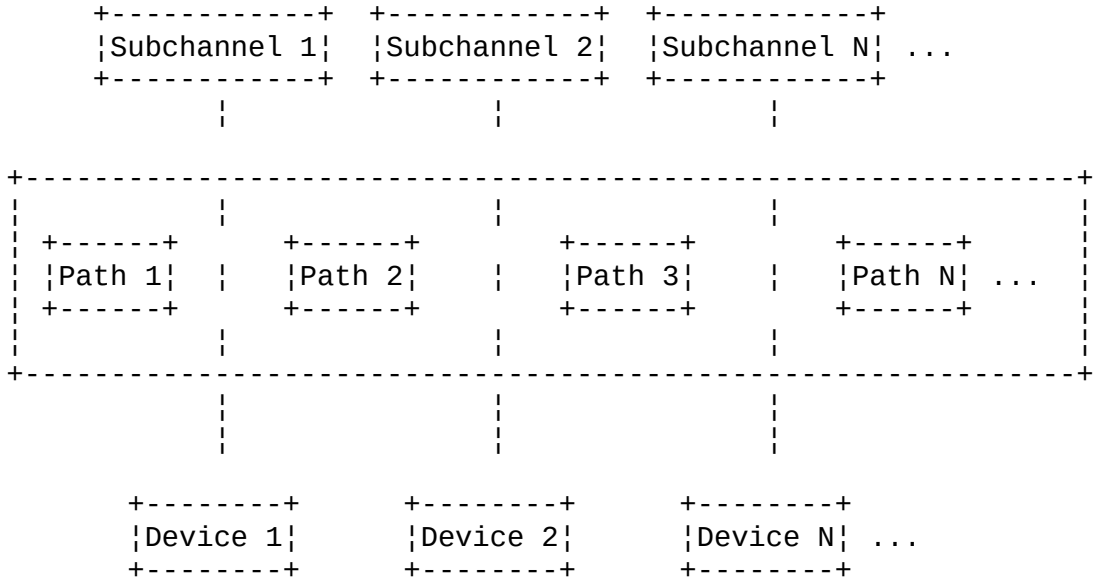


In the AIX/370 system, each subchannel is associated with
specific channel path. It may use any available device

Figure C-3. AIX/370 Path Management

370-XA

Channel Subsystem



In the 370-XA system, each subchannel is associated with a specific device. It may use any available path.

Figure C-4. 370-XA Path Management

The ability of a 370-XA system to use multiple paths and to share paths among devices is part of the path management facilities. Dynamic path reconnection is the ability of an operation to begin on a path and to disconnect, and reconnect to any available path at a later time.

In 370-XA mode, a device can be connected to multiple control units, and each control unit can be connected to a subchannel by multiple channel paths.

Note: The device is addressed by a single logical address (even though multiple paths through separate control units actually exist).

In AIX/370 mode, the device in the above configuration would have two separate channels and be in two separate channel sets. It could be accessed by two separate (channel/path) addresses.

Each subchannel has an associated subchannel number, from 0 to 64k-1. All of the 370-XA I/O instructions reference the subchannel number as an instruction argument.

Each channel path is assigned a number from 0 to 255. Each device is assigned a device number from 0 to 64k-1. The device number is used by most CP commands.

The new 370-XA I/O instructions are:

(*) - Similiar in function to AIX/370 counterparts.

Clear Subchannel (*)
Halt Subchannel (*)
Modify Subchannel
Reset Channel Path (*)
Resume Subchannel
Set Address Limit
Set Channel Monitor
Start Subchannel (*)
Store Channel Path Status
Store Channel Report Word
Store Subchannel (*)
Test Pending Interruption
Test Subchannel (*)

I/O is initiated using the Start Subchannel instruction, whose arguments are the address of an operation request block (ORB) and a subchannel number. The ORB structure contains:

CAW - address of the channel progra
Interruption parameter
Flags.

The interruption parameter is a word which may be used by the upper layers of software to identify the operation and is presented as part of the channel interrupt.

The flags set/reset various features of 370-XA mode I/O such as:

Subchannel ke
Suspend contro
Format control - 24/31 bit CCW forma
Prefetch contro
Initial status interrup
Address limit checkin
Suppress suspended interruptio
Logical-path mas
Incorrect length suppression mode

I/O completion is typically signified by the primary-interruption condition which causes a subchannel status word (SCSW) to be stored in the subchannel. Alternatively, the **Test Pending Interruption** instruction can be used to detect completion when interrupts are disabled.

The SCSW is part of the interrupt response block (IRB) which is retrieved from the subchannel using the **Test Subchannel** instruction. The IRB contains the following fields:

Subchannel Status Word (SCSW)
Extended-status Word
Extended-control Word.

For each subchannel, there exists an entity known as the subchannel information block (SCHIB). The SCHIB contains the following:

Path Management Word
Subchannel Status Word

Model-dependent Area.

The SCHIB is read using the Store Subchannel instruction and can be modified and re-programmed using the **Modify Subchannel** instructions. Each subchannel can be assigned a specific I/O interrupt level which can be selectively masked using control register 6.

The **Set Address Limit** instruction is a method of protecting main memory from I/O access. This instruction allows a physical address range to be protected against I/O write access and is system-wide.

370-XA mode supports suspend-resume operations of CCWs within a channel program and also supports a sophisticated measurement facility for monitoring I/O.

When a change occurs to the device configuration at the CP level, the virtual machine receives a special interrupt signifying that a channel report word (CRW) is available. The **Store Channel Report Word** instruction is used to obtain this report.

For complete information consult the following documents:

IBM System/370 Principles of Operation
IBM System/370 Extended Architecture Principles of Operation
Device specific literature

C.3.2 AIX/370 Device Drivers

The device driver typically prepares one or more channel program segments in static memory during driver initialization.

Some channel programs are skeletal and should be customized in accordance with each I/O request. For example, a disk driver tailors each read/write channel program by specifying the track/sector combination. Other devices, such as an Ethernet driver may use static channel programs.

Subtopics
C.3.2.1 AIX/370 I/O Subroutines

C.3.2.1 AIX/370 I/O Subroutines

The kernel routines to support the S/370 channel subsystem are described in the following sections. Some of the routines are in S/370 assembler and are really accessed through routine, machdep. The names used in this document are the ones from the sample driver, not the real function. The typedefs are shown in the section on AIX/370 I/O Subroutines as well as in the **usr/include/sys/b370** or **usr/include/sys /XA370** directories.

mkccw

Routine **mkccw** will fill in a structure of type **ccs_t** with the parameters supplied.

```
mkccw(cp, cmdcode, lastflag, nbyte, buffp)
register ccw_t *cp   /* pointer to ccw structure */
char cmdcode;   /* CCW command code */
char lastflag;   /* CCW flags (CC, CD, SLI etc) */
int nbyte;   /* io byte count */
register paddr_t buffp;   /* pointer to buffer */
```

build_ccw

Routine **build_ccw** builds **ccw** lists and an IDAW for data buffers that cross 2k page boundaries or if not will invoke **mkccw**.

```
build_ccw (cp, cmdcode, lastflag, nbyte, buffp, addr)
register ccw_t *cp;   /* pointer to ccw structure */
char cmdcode;   /* CCW command code */
char lastflag;   /* CCW flags (CC, CD, SLI etc) */
int **addr;   /* io byte count */
register paddr_t buffp;   /* pointer to buffer */
register int addr;   /* build area for IDAL */
```

redrive_dev(dev)
redrive_cu(dev)
redrive_sio(dev)

These routines will redrive the device, control unit, or channel for XA370 type of channel protocol. See 370 Principles of Operation.

physiolock

This routine will Lock a piece of physical storage for I/O given the starting page boundary address and the number of pages.

```
physiolock(base, count)
caddr_t base;   /* starting page address */
int count;   /* number of pages */
```

physiounlock

This routine will Unlock a piece of physical storage for I/O given the starting page boundary address and the number of pages. If rw is **B_READ** then the page must be written before unlocking.

```
physiounlock(base, count, rw)
caddr_t base;   /* starting page address */
int count, rw;   /* number of pages, use flage */
```

sio

This routine will issue the S/370 SIO instruction and will return the condition code and the **CSW** if stored.

```
sio(dev, ccw, csw)
```

```
    int dev;   /* hex CUU address of the device */
    ccw_t *ccw;   /* address of the CCW list */
    csw_t *csw;   /* address of CSW data for caller */
```

tsio

This routine will do a test I/O to clear the path and then do **sio**. It will then return the condition code and the **CSW** if stored.

```
    tsio(dev, ccw, csw)
    int dev;   /* hex CCU address of the device */
    ccw_t *ccw;   /* address of the CCW list */
    csw_t *csw;   /* address of CSW data for caller */
```

syncio

This routine will issue the **SIO** and then spin on a **TIO** until ending status is received. The caller should NOT do a sleep because the devices' **xxxintr** routine will NOT be called. When complete, will return the condition code and the **CSW**. It is possible to get interrupts.

```
    syncio(dev, ccw, csw)
    int dev;   /* hex CUU address of the device */
    ccw_t *ccw;   /* address of the ccw list */
    csw_t *csw;   /* address of CSW data for caller */
```

hdv

This routine will issue a **HIO** to terminate a pending **SIO** and will return the condition code and the **CSW**.

```
    hdv(dev, csw).
    int dev;   /* hex CUU address of the device */
    csw_t *csw;   /* address of CSW data for caller */
```

tio

This routine will issue a **TIO** to obtain device status and will return the condition code and the **CSW**.

```
    tio(dev,csw)
    int dev;   /* hex CUU address of the device */
    csw_t *csw;   /* address of CSW data for caller */
```

svc76

This routine passes the record to **VM** for **EREP** logging.

```
    svc76(size, obr ] mdr)
    int size;   /* length of the record */
    *obr ] *mdr   /* address of the data */
```

useracc

This routine access the S/370 protect keys to determine if the address is valid for this user. If not, OK return 0.

```
    useracc(iobase, count, type)
    caddr_t iobase;   /* start addr of user buffer */
    unsigned count;   /* number of bytes to check */
    int type;   /* read or write access flag */
```

A unique requirement to AIX/370 device drivers is that, because AIX/370 runs as a virtual machine, other virtual machines may be sharing the physical devices; thus, although a device driver knows when it has no

outstanding requests, the device or a path to the device may be busy. A certain amount of retry is built into the **sio** and **syncio** routines to account for the possibility of device-in-use status.

A driver receives interrupts from its controlled device, by registering the device address and an interrupt handling routine. Interrupts generated by the specified device cause the driver's interrupt routine to be called.

The status passed to the driver's interrupt routine contains the CSW, which the driver uses to obtain completion status, residual count and other device status.

Devices can only perform a single operation at a time; it is the responsibility of the device driver to supply the device with activities with the goal of keeping the device busy. The driver typically achieves high device utilization by starting a device operation as soon as the device completes an operation. The interrupt handler is a natural place for this intelligence, since the interrupt routine is called for device operation completion.

A device driver accepts requests from the kernel and initiates device activity if the device is idle, or it queues the request until the device becomes inactive.

Some device types need to bring attention to the kernel that an activity was performed; the following are examples of such activities: a keystroke is generated at a terminal, a device goes on-line, a packet is received over a communications medium. The device generates an attention interrupt in order to communicate this change of status to the driver.

C.4 Types of Device Drivers

In the AIX kernel framework, the two general types of device drivers are character and block. In addition, the two special types of device drivers are TTY and network. A character device driver sends and receives ***raw*** input from and to AIX applications that have issued system calls, meaning that the data is not manipulated by the kernel before it arrives at the character device driver. A block device driver uses the kernel buffer cache so that I/O appears asynchronous to the state of the hardware. A TTY device driver is a type of character device driver that is designed to interface with teletype terminals, usually connected through an RS-232 interface. A TTY driver uses line discipline routines inside the kernel to buffer terminal I/O. A network device driver is accessed by user programs through sockets. The AIX kernel provides protocol functions so that network device drivers can interface to Ethernet, Token-Ring, X.25, and other physical layer protocols. Figure C-5 presents an overview of device driver types.

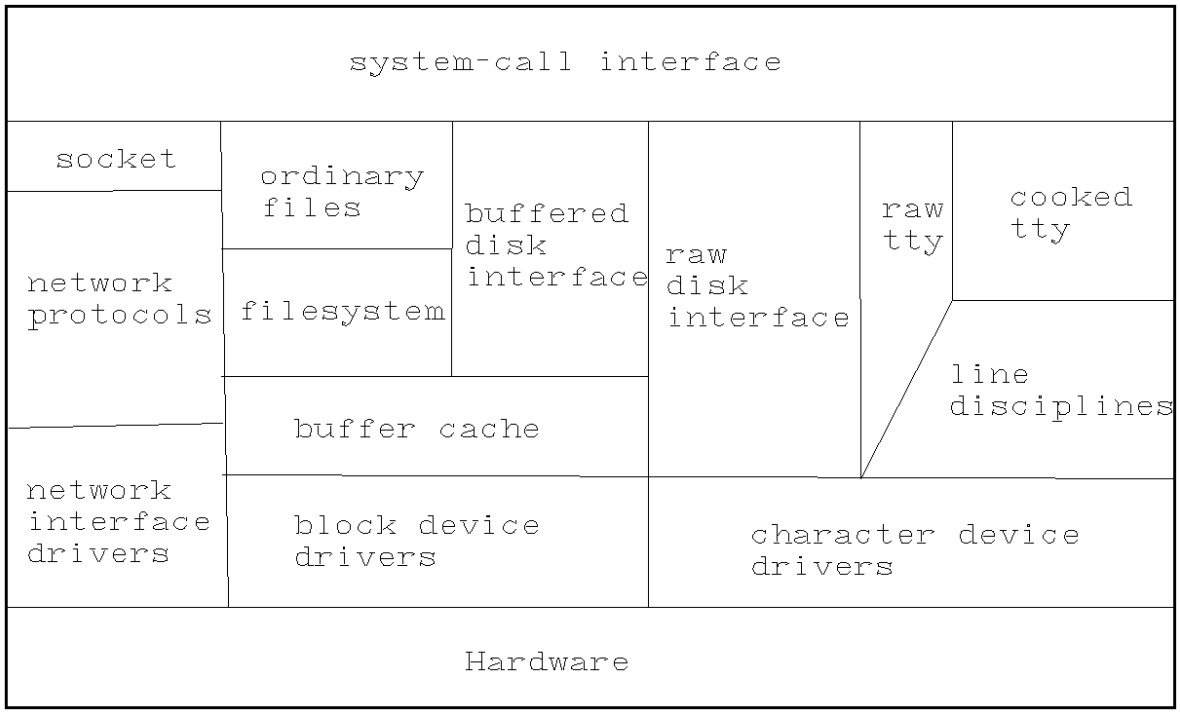


Figure C-5. Overview of Device Driver Types

There is no one-to-one correspondence between system call parameters and the parameters passed to device driver entry points. In character and TTY device drivers as well as for raw interface to network and block device drivers, most system call parameters are passed to the device driver in the user structure. The extended parameter, used in **openx**, **readx**, **writex**, and **ioctlx** system calls, is passed as a parameter to the device driver's entry point. Block and network device drivers are passed information about the calling application in the buffer cache and the **mbuf** chain respectively.

Subtopics
C.4.1 Basic Device Driver Template

C.4.1 Basic Device Driver Template

Every type of device driver has a basic template of entry points: *dd***init**, *dd***reset**, *dd***open**, *dd***close**, and *dd***intr**. This section details these entry points in terms of their interface to the kernel and their device-specific attributes. If the entry point is specific to AIX PS/2 or AIX/370, it is so indicated; otherwise, the entry point is common to both systems.

Subtopics
C.4.1.1 Entry Points
C.4.1.2 Basic Template Kernel Subroutines and Data Structures
C.4.1.3 devdata Data Structure

C.4.1.1 Entry Points

---

ddinit (**PS/2**)

```
ddinit (devno)
dev_t devno;
```

The ***dd*init** entry point is used to configure the device driver. This entry point is called after the virtual space and **malloc** arena are established and before any processes have been spawned by the main routine of the AIX kernel. Interrupts are disabled at this point and may not be enabled. Each device driver's ***dd*init** is called once.

Parameter:

devno Contains the device major number. Use the **major** macro defined in "Determining Major and Minor Numbers" in topic C.6.6 to obtain the major number from the device type.

For autoconfigured devices, ***dd*init** is the driver entry point where autoconfiguration takes place. Therefore, the ***dd*init** entry point of autoconfigured device drivers should perform the following functions:

Check for the presence of the associated hardware by calling th **devexist** kernel subroutine

Obtain the device's POS information from the **devdata** structure

Perform the necessary hardware initialization, before the device ca be used

Call the **DEV_INSTALL**, **CDEV_INSTALL**, and **BDEV_INSTALL** routines to install the device driver entry points into the kernel's **devsw** table

Call the **intrattach** routine to install the device driver interrupt handler so that the first level interrupt handler will call it

Perform data structure allocation and initialization

A sample ***dd*init** routine for a block device that has raw device entry points follows:

```
extern struct devdata devdata;
struct iobuf sampbuf;

sampinit ( dev )
dev_t dev;
{
    int maj;
    int slot;
    int samp_levl;            /* interrupt level */

      maj = major ( dev );

      if ( ( slot = devexist(cardid, maj, 1, 0 ) ) != -1)
      {
```

```
                DEV_INSTALL ( maj, sampinit, nulldev,
                  sampopen, sampclose, sampintr,
                  ISNOTATTY );
              BDEV_INSTALL ( maj, sampstrat, nulldev,
                  &sampbuf );
              CDEV_INSTALL ( maj, sampread, sampwrite,
                  sampioctl, nulldev, notty );

              /* get the interrupt level from POS data - position
                 in POS varies by adapter.  Some adapters may
                 not have the interrupt level recorded in the POS */
              samp_levl = devdata[slot].pd_pos3;

              intrattach ( sampintr, samp_levl, SPLBLKIO );
          }
      }
```

***dd*init (AIX/370)**

The ***dd*init** routine is called for each device in the **gensw** table with the following arguments:

```
***dd*init(*devs*, *units*)
struct dev_unit **devs*;
int units;
```

Parameters:

devs Points to a table of *units* device structures.

units Number of configured *units*.

The **dev_unit** structure, which is the device information table, is defined in **/usr/include/sys/conf.h** as follows:

```
  struct dev_unit{

    char dvu_name[9];          /*configured name of this particular device*/
    char dvu_type[9];          /*configured type of this particular device*/
    int dvu_units;             /*number of sub-units associated with device*/
    char *dvu_feat;            /*configured features/options for device*/
    long dvu_info[N_DVU_INF];  /*address information for this device*/
  };
```

When the AIX/370 is rebooted (IPL'ed), one of the first operations performed is the auto-configuration process. Autoconfiguration consists of calling each device's ***dd*init** routine once. During the generation of the AIX kernel, a table of configured devices, **gensw**, is composed. The configuration is built by the **config** program as it parses **/etc/master** and **/etc/system** files. The resultant table becomes part of **conf.c**, which is compiled and linked with the AIX kernel.
An example table is shown below.

```
  /*
   * System configuration summary
```

```
 */
struct gen_sw gensw[] = {
/*          type          blk    chr    units    table */
            {"sysdevs",   -1,    23,    1,       0 },
            {"mt",        -1,    1,     1,       mtdevs },
            {"console",   -1,    7,     1,       cndevs },
            {"net",       9,     -1,    1,       0 },
            {"nfs",       39,    -1,    1,       0 },
            {"memory",    8,     8,     1,       0 },
            {"dkfba",     0,     0,     17,      fbadevs },
            {"dkckd",     4,     4,     12,      ckddevs },
            {"lp",        -1,    13,    2,       lpdevs },
            {"lcs",       -1,    15,    2,       lcsdevs },
            {"osm",       -1,    16,    1,       0 },
            {"tty",       -1,    17,    1,       0 },
            {"pts",       -1,    2,     1,       0 },
            {"ptc",       -1,    18,    1,       0 },
            {"mon",       -1,    19,    1,       mondevs },
            {"ldsf",      -1,    20,    1,       ldsfdevs },
            {"pun",       -1,    21,    4,       pundevs },
            {"rdr",       -1,    22,    3,       rdrdevs },
            {"nty",       -1,    25,    1,       0 },
            {"vctc",      -1,    26,    3,       vctcdevs }
};
```

The **config** program will generate a **devs** table prefixed by the name of the driver for each entry in the system file. For example, for the **fba** driver, the following has been extracted from **conf.c**:

```
/*
 * Configuration for fba driver
 */
struct dev_unit fbadevs[] = {
/*          name           type      units    parms    address */
            { "IX370S",    "3370",   1,       "o",     { 0x340, } },
            { "IXVOL0",    "3370",   1,       "o",     { 0x341, } },
            { "IXVOL1",    "3372",   1,       "o",     { 0x348, } },
            { "IXVOL2",    "3372",   1,       "o",     { 0x349, } },
            { "IXVOL3",    "3372",   1,       "o",     { 0x34a, } },
            { "IXVOL4",    "3372",   1,       "o",     { 0x34b, } },
            { "IXVOL5",    "3372",   1,       "o",     { 0x342, } },
            { "IXVOL6",    "3372",   1,       "o",     { 0x34c, } },
            { "IXVOL7",    "3372",   1,       "o",     { 0x34d, } },
            { "IXVOL8",    "3372",   1,       "o",     { 0x370, } },
            { "IXVOL9",    "3372",   1,       "o",     { 0x350, } },
            { "IXVOLA",    "3372",   1,       "o",     { 0x351, } },
            { "IXVOLB",    "3372",   1,       "o",     { 0x352, } },
            { "IXVOLC",    "3372",   1,       "o",     { 0x353, } },
            { "IXVOLD",    "3372",   1,       "o",     { 0x354, } },
            { "IXVOLE",    "3370",   1,       "o",     { 0x343, } },
            { "IXVOLF",    "3370",   1,       "o",     { 0x344, } },
};
int         fba_cnt = 17;
```

Each line in the **fbadevs** structure was built from a stanza in the system file such as:

```
            IXCKD0:
                  driver = 3370
```

```
address = 340
features = o
```

The *dd***init** entry point should perform the following functions:

Insert the address of the drivers **mbstrategy** routine into the **mbdevsw** if this is a block driver supporting a multi-block strategy interface.
Initialize prototype channel programs by allocating storage for th actual CCWs comprising the required channel programs, noting required alignment (IDAWs - fullword, CCWs doubleword). Build a list of linked CCWs with appropriate field initializations.
Perform driver specific static data structure initialization o allocation of dynamic memory requirements.
Perform driver attach operation (optional) which performs any actua additional initialization activity as required by a specific driver.

Note: The use of a separate **attach** routine is optional.

For both the PS/2 and the AIX/370 network device drivers, the *dd***init** entry point additionally initializes an **ifnet** structure, as seen in "Network Device Drivers" in topic C.4.5, and then attaches it to the kernel using the **if_attach** kernel subroutine. Each network port associated with the device driver must be attached with the **if_attach** routine. The following is an example of how to initialize the **ifnet** structure:

```
struct ifnet eth.ifs[MAXETHUNITS];

ethattach(unit)
   int unit;
{

struct ifnet *ifp = eth_ifs[unit];
ifp -> if_units = unit;
ifp -> if_name = "eth";
ifp -> if_mtu = ETHERMTU;
ifp -> if_ioctl = eth_ipc_ioctl;
ifp -> if_output = eth_ipc_output;
ifp -> if_flags = IFF_BROADCAST|IFF.NOTRAILERS
             IFF_ETHERNET|IFF_IEEE;
if_attach(ifp);
}
```

---

ddreset (**AIX PS/2**)

*dd***reset** ()

The *dd***reset** entry point of an AIX device driver is called by the kernel when it is shutting the system down. This is an optional entry point and may be used to perform device-specific shutdown procedures such as parking disk heads.

The *dd***reset** entry point has no parameters.

The following *dd***reset** routine is taken from the diskette device driver:

```
fdpark()
```

```
{
        int i;
        if( fdtab.ib_active & 0xff) {
            printf("fdpark: motor(s) not killed, i/o still pending\n);
            return;
}
/*Reset both the controllers*/
fd.dor [0]=0;
iooutb (SYSTEM_BASE + DOR, 0);
fd.dor [1]=0;
iooutb (ADAPTER_BASE + DOR, 0);

fd.nsec=0;
fd.state=RSET;
```

---

ddopen

```
dd open (dev, flag, ext)
dev_t dev;
int flag;
caddr_t ext;
```

The *dd***open** entry point of an AIX device driver is called by the kernel when a program issues an **open** or **create** system call.

Parameters:

dev Contains the device's major and minor number. Use the **major** and **minor** macros to obtain major and minor numbers from a device type.

flag This is the value of the **oflag** argument passed to the **open** system call. It is one of the following values, as defined in the **/usr/include/sys/file.h** or **/usr/include/sys/fcntl.h** header files:

O_RDONLY	Device is being opened for reading only by ensuring that it is mounted and/or powered on.
O_WRONLY	Device is opened for writing only.
O_RDWR	Device is opened for reading and writing.
O_NDELAY	Request a non-blocking open, that is, allow the process to exit the **open** routine before waiting for the device to become ready. This is typically used by tty drivers only.
FDEVHANDLE	Indicates an open for a multiplexed device to map the multiplexed name of a subchannel.
FAUTOCONF	Indicates a kernel-initiated open at system startup for autoconfiguration of the root, pipe, swap, and dump devices. Support for **FAUTOCONF** is usually only needed in disk drivers. A driver declares its ability to support **FAUTOCONF** with the **DEV_AUTOCONF** flag of the **DEV_INSTALL** routine.

ext The value of the **ext** parameter which is passed to the **openx** (extended **open**) system call.

The main purpose of the *dd***open** entry point is to validate the minor number, initialize data structures, and condition the device for use.

The *dd***open** entry point can indicate an error condition to the application program by storing a nonzero error code in the **u.u_error** field of the user block, causing the **open** system call to fail. In this case, the system call returns a value of -1 and the error code is available to the application in the **errno** external variable. The error code used should be one of the values defined in the **/usr/include/sys/errno.h** header file.

After the user process opens any special file, the kernel converts the special file name into an inode pointer by lexing through the full path name of the special file. When the kernel determines that the process is attempting to open a special file, it determines the special file's inode and then calls the **open** routine of the device driver. If the special file is multiplexed, the kernel stops lexing after the multiplexed inode is encountered and calls the **open** routine of the device driver twice - once to determine the multiplexed subchannel and again to actually open the subchannel.

During the first open, the **flag** parameter is set to **FDEVHANDLE**. The **open** routine inquires **u.u_dirp**, a pointer to the path name extension following the special file name used in the **open** system call. For example, if you open the **/dev/hft** file, **u.u_dirp** is a pointer to an empty (NULL) string. If you open the **/dev/hft/01** file, **u.u_dirp** points to the ASCII string: **01**. If **u.u_dirp** is a pointer to an empty string, the driver places a unique channel number in the **u.u_mpxchan** field. If the **u.u_dirp** field is non-NULL, the driver places the requested channel number, converted from an ASCII string to binary, in the **u.u_mpxchan** field. The **open** routine returns immediately once the device driver places a value into the **u.u_mpxchan** field.

Note: It is the responsibility of the device driver writer to maintain a list of used and unused multiplexed channel numbers.

The AIX kernel immediately calls the **open** routine again, this time with the flag set to the value of **oflag** from the **open** system call. It also accesses the process's channel number in the **u.u_mpxchan** field and performs the regular open functions.

The *dd***read**, *dd***write**, *dd***ioctl** and *dd***select** entry points of the driver can also access the channel number in the **u.u_mpxchan** field.

If a character device uses line discipline routines, then *dd***open** should initialize the **tty** structure and then call the **l_open** line discipline routine. The **t_line** field of the **tty** structure is used to index into the line discipline table. Currently, 0 is for **tty** and 1 is for printer.

Many character devices, such as printers and plotters, should be opened by only one process at a time. The *dd***open** entry point can enforce this by maintaining a static flag variable, which is set to 1 if the device is open and 0 if not. A device driver that requires special software downloading or initialization, such as an intelligent controller, should return ENXIO if the device has not yet been initialized. Each time it is called, *dd***open** checks the value of the flag and, if it is other than 0, sets **u.u_error** to **EBUSY**; otherwise, the *dd***open** entry point sets the flag and returns normally. The *dd***close** entry point later clears the flag when the device is closed. For more information on the standard errors returned by **open**, see "open, openx, creat" in topic 1.2.199.

Most block devices can be used by several processes at once, and the device driver should not usually enforce a single-user restriction.

Security measures are usually performed in *dd***open**. Some devices should allow only one user to access at a time. This can be performed by comparing either the effective or the real user ID of the request to open to that of the process that already has the device opened.

---

ddclose

```
dd close (dev, flag, ext)
dev_t dev;
int flag;
caddr_t ext;
```

The *dd***close** entry point resets the device to a known state, resets the device controller to prevent it from generating any more interrupts until it is opened again, flushes any outstanding I/O requests, and on the last close of the device, unmounts the device.

Parameters:

dev Contains the device's major and minor number. Use the **major** and **minor** macros to determine the major and minor number of the device type.

flag The **oflag** argument passed to the last **open** system call. See the **/usr/include/sys/file.h** header file for a complete definition of the bits in the **flag** parameter word.

ext The value of the **ext** parameter passed to the **closex** (extended **close**) system call.

The *dd***close** entry point is only entered on the final close of a minor number.

This entry point should always succeed, that is, the **u.u_error** field should never be set to a nonzero value in *dd***close**.

---

ddintr (**AIX PS/2**)

```
dd intr (vec_num)
int vec_num;
```

The *dd***intr** entry point is called by the kernel when a device issues an interrupt.

Parameter:

vec_num Is an integer that specifies the interrupt vector number.

An interrupt typically signals completion of a data transfer. The *dd***intr** routine must determine the appropriate action, for example by taking the received character and placing it in the input buffer, or by removing the

next character from the buffer and starting the transmission. The ***dd*intr** entry point also checks error state and initiates the next I/O operation.

Interrupts on the PS/2 are vectored through two 8259 Programmable Interrupt Controllers (PICs). Each PIC has eight individually maskable interrupt request levels. Because the second PIC is cascaded off one of the interrupt request levels associated with the first PIC, there are only 15 interrupt request levels available. The interrupt request level associated with a given PS/2 device is either predetermined by the hardware or user-configured via the PS/2 reference disk. Multiple devices can share an interrupt request level. Hence, device drivers must be prepared to field spurious interrupts. All PS/2 devices have an interrupt-asserted status bit that the device driver can query to determine if it owns the interrupt. Device drivers must clear the interrupt from the device prior to returning from the ***dd*intr** entry point when required by the hardware.

Block device interrupt handlers are responsible for freeing each block after it has been successfully transferred to the storage devices. Use the **iodone** kernel subroutine to free disk buffers.

---

ddintr (**AIX/370**)

```
ddintr (da, csw, unit)
ioaddr_t da;
csw_t csw;
int unit;
```

Parameters:

da Contains the device address

csw Is the completion channel status word

unit Indicates the driver unit number associated with this device

On the AIX/370 all devices generate an I/O interrupt on level 4. When a device generates an interrupt, the CPU stores the current processor status word (PSW) in the level 4 old PSW indicator and loads a new PSW from the level 4 new PSW location. This causes the CPU to begin executing the kernels generic interrupt handler. During the processing of an I/O interrupt the CPU will not take another interrupt.

For the purpose of the generic interrupt handler (the primary interrupt handler) the CPU must save the context of the currently running process and then construct a trap block to pass to the trap handling code. This trap block will contain all of the volatile information which accrues when an interrupt occurs. This includes the CSW stored in page zero, the device address, as well as the programmable interrupt word.

The code in trap will invoke the secondary interrupt handler which is the driver specific handler previously setup via the **addiohdlr** routine based on the device address. The kernel will dispose of any interrupt for which a second-level interrupt handler (SLIH) has not been established.

C.4.1.2 Basic Template Kernel Subroutines and Data Structures

The following kernel subroutines define the virtual interrupt handler to AIX.

---

intrattach (AIX PS/2)

```
#include <i386/intr86.h>

intrattach (func, level, splmask)
int (*func)();
int level, splmask;
```

The **intrattach** kernel subroutine binds an interrupt request level with an interrupt priority.

Parameters:

func Specifies interrupt handler

level Specifies interrupt request level

splmask Defines an interrupt's priority.

Interrupts are masked by **splmask** in the following manner:

1. When a process explicitly masks an interrupt in a device driver, any interrupt request levels bound to **splmask**, including the interrupt request level specified by **level**, do not cause the process to be interrupted.

2. While in the interrupt handler, all interrupt request levels bound to **splmask** do not cause the interrupt handler to be pre-empted.

Callers of **intrattach** must ensure that an appropriate mask is assigned to an interrupt request level. For example, devices that are not time critical, such as tape drives, should not be bound to the **splmask** value **SPL_HIGH**. In addition, note that low priority devices can share interrupt request levels with high priority devices. Therefore, low priority devices mask interrupts for high priority devices. When configuring a PS/2 with the reference disk, either assign unique interrupt request levels to configured devices or attempt to group devices of the same type, such as fixed disks or serial communication devices, on the same interrupt request level. PS/2 microchannel supports sharing interrupt levels.

The appropriate values of **splmask** are as follows:

SPL_HIGH No interupts at this level

SPL_CLKONLY Mask out all but the clock

SPL_IMP Mask out network devices

SPL_BLKIO Mask out devices which use the buffer pool tape or disk

SPL_CLIST Mask out devices which deal with **clist** structures

SPL_KEYBRD Mask out the console keyboard

SPL_MISC Mask out miscellaneous devices

SPL_SWINTR Mask out VAX-simulated software interrupts

---

intrdetach (AIX PS/2)

```
intrdetach (func, level)
int (*func)();
int level;
```

The **intrdetach** kernel subroutine detaches a currently active interrupt handler from the operating system list. Valid interrupts are from 0-15.

Parameters:

func Specifies interrupt handler

level Specifies interrupt request level

---

devexist (AIX PS/2)

```
int devexist (cardid, d_major, flags, start_slot)
unsigned short cardid, d_major, flags;
int start_slot;
```

The **devexist** kernel subroutine checks to see if the adapter used by a device driver exists in the PS/2. Returns either the slot number (**start_slot** to NIOSLOTS()) or -1.

Parameters:

cardid 16-bit adapter ID which is contained in the first two bytes of POS data.

d_major The major number of the device.

flags Device flags that you would like placed into **devdata[slot].pd_flags**. This parameter could be used to identify logical adapter numbers in device drivers that support multiple adapters of the same **cardid**.

start_slot The slot number used to start searching for the adapter (0-NIOSLOTS()).

An autoconfigured device driver does not install itself into the device switch table unless it determines that the hardware it needs to run has been placed into the system.

Note: The **devexist** routine may have to be called multiple times for drivers that support many adapters of the same adapter ID or alternate adapter IDs.

C.4.1.3 devdata Data Structure

The AIX kernel contains a POS structure that contains all the POS information for a given hardware device. The name of the kernel structure is called **devdata**. The **devdata** structure contains the adapter ID, 4 bytes of option select data, the device major number and flags.

Autoconfigured device drivers use **devdata** during the *dd***init** routine to determine whether or not they should install their entry points into the device switch table.

The **devdata** structure is defined as follows:

```
struct devdata {
        unsigned char pd_pos0;   /* card ID */
        unsigned char pd_pos1;   /* card ID */
        unsigned char pd_pos2;   /* pos reg 1 */
        unsigned char pd_pos3;   /* pos reg 2 */
        unsigned char pd_pos4;   /* pos reg 3 */
        unsigned char pd_pos5;   /* pos reg 4 */
        unsigned char pd_pos6;   /* pos reg 5 */
        unsigned char pd_pos7;   /* pos reg 6 */
        unsigned char pd_major;  /* major number */
        unsigned char pd_flags;  /* device flags */
};
```

DEV_INSTALL

```
DEV_INSTALL (maj, init, reset, open, close, intr, flags)
int maj;
int init;
int reset;
int open;
int close;
int intr;
unsigned int flags;
```

The **DEV_INSTALL** kernel subroutine installs the common entry points of a device driver so that the driver can be called.

Parameters:

maj The major number of the device.

init Pointer to *dd***init** or **nulldev**.

reset Pointer to *dd***reset** or **nulldev**. Routine that is called on system shutdown.

open Pointer to *dd***open** or **nulldev**. Routine that is called for the **open** system call.

close Pointer to *dd***close** or **nulldev**. Routine that is called for the **close** system call.

intr Pointer to *dd***intr** or **nulldev**. Not used by the system.

flags One or more of the following (ORed together as necessary):

ISNOTATTY	Default for most device drivers.
ISATTY	Device driver is a **tty** type device.
ISMPX	Device driver supports a multiplex file.
DV_ATDMA	Device may use AT-style DMA transfers (for AT-compatible drivers). Not generally used.
DV_AUTOCONF	Device will attempt auto-configuration.
DV_TAPE	Device driver is for tapes. A tape device that does not support read ahead or delayed writes.
DV_MINBLK	Minimal block device; no seeks or mounts allowed. Minimal block devices have the benefits of buffer caching and asynchronous I/O but no file systems may be placed on the device.

CDEV_INSTALL

CDEV_INSTALL (**maj, read, write, ioctl, select, tty**)

The **CDEV_INSTALL** kernel subroutine installs character device entry points.

Parameters:

maj The major number of the device.

read Pointer to *dd***read** or **nodev**. Routine that is called to handle the **read** system call.

write Pointer to *dd***write** or **nodev**. Routine that is called to handle the **write** system call.

ioctl Pointer to *dd***ioctl** or **nodev**. Routine that is called to handle the **ioctl** system call.

select Pointer to *dd***select** or **seltrue**. Routine that is called to handle the **select** system call.

tty Pointer to a routine that returns the address of the **tty** structure associated with a particular minor number or **notty**.

BDEV_INSTALL

BDEV_INSTALL (**maj, strat, dump, tab**)

The **BDEV_INSTALL** kernel subroutine installs block device entry points.

Parameters:

maj The major number of the device.

strat Pointer to *dd***strategy** or **nostrat**. Called by the kernel to schedule I/O.

dump Pointer to *dd***dump** or **nulldev**. Called by the kernel for dump I/O.

tab Pointer to the block device table (**struct iobuf**). Not currently used.

---

if_attach

```
if_attach (ifp)
struct ifnet *ifp;
```

Parameter:

ifp A pointer to a device-specific **ifnet** structure.

The **if_attach** kernel subroutine attaches a network interface to the list of active interfaces. For network device drivers only.

C.4.2 Character Device Drivers

Character device drivers offer a direct path between the user and the hardware device. The interface to character device drivers is less structured, and thus more flexible, than the interface to block devices. Character devices include the keyboard, displays, printers, special purpose hardware (such as a mouse), and software pseudo-devices.

Character device drivers can be designed to provide controlled access to low-level facilities of the system that are not necessarily associated with true I/O devices. AIX provides several special-purpose device drivers:

/dev/null Discards output written to it and indicates an end-of-file condition when read.

/dev/trace Records and returns data when tracing programs.

/dev/error Records and returns system errors.

/dev/mem Provides access to physical memory.

/dev/kmem Provides access to kernel memory.

/dev/osm Records kernel message logged to the console.

***dd*attach**

The ***dd*attach** operation can be invoked from the **init** routine or from the **open** routine and it is performed only once. The function of the ***dd*attach** routine varies from driver to driver but typically includes one of the following:

Initialize network interface and call **if_attach** (network drivers)
Determine device characteristics and initialize driver privat information
Read the VTOC informatio
Print some console information about the device (address
Install the interrupt handler **addiohdlr**).

An interrupt handler must be setup for each unit. The actual call would be performed as follows:

addiohdlr (**ioaddr, inthdlr, unit**);

Parameters:

ioaddr The physical device address

inthdlr The interrupt service routine

unit The device unit number or other argument to receive with the interrupt from this device

Note: Typically, there is only one interrupt service routine for all devices serviced by a single device driver.

Subtopics
C.4.2.1 Character Device Driver Data Structures
C.4.2.2 Character Device Driver Entry Points

C.4.2.1 Character Device Driver Data Structures

Character device drivers, as well as TTY device drivers and the raw interfaces to block and network device drivers, access user data through the user block. The user block contains some of the information that the kernel keeps about the user process. The entry points that can access the user block are *dd***open**, *dd***close**, *dd***ioctl**, *dd***read**, *dd***write**, and *dd***select**.

The user block can be accessed with the construct **u.field**, where **field** is a valid field name of **struct user** as defined in the **/usr/include/sys/user.h** header file. The name of each of the fields in this structure begins with **u_** so user block references take the form **u.u_name**.

Warning: Do not modify any user block fields or any other kernel structure fields that are not explicitly mentioned in this appendix. Otherwise, unpredictable results may occur.

Some of the frequently used fields of the user block are:

u.u_error If the device driver stores a nonzero error code in this field, then it is passed to the application program in **errno** and a -1 is returned by the system call. The valid error codes are defined in the **/usr/include/sys/errno.h** header file. See Appendix A, "Error Codes" in topic A.0 for a description of these error codes.

u.u_fmode This is the **open** flag or file parameter word for the open file descriptor associated with the device. It is the same value that is passed to the *dd***open** and *dd***ioctl** entry points as the **flag** parameter.

u.u_qsav If a signal is received while a process is in kernel mode, a **longjmp** is normally performed, passing control to the address saved in this field. By default, **u.u_qsav** points to a routine that sets **u.u_error** (**errno**) to EINTR and returns the value -1 to the user process, indicating that the system call was interrupted by a signal. This action can be overridden under certain circumstances. See "Process Suspension and Timing" in topic C.6.2 for more information.

Fields used by the *dd***read** and *dd***write** entry points are:

u.u_base The address of the beginning of the data buffer from and to which data is transferred. The value of **u.u_base** is incremented after each character is transferred to or from it.

u.u_count The byte count given to the **read** or **write** system call. The value of **u.u_count** is decremented after each character is transferred to or from **u.u_base**.

u.u_icount The initial byte count given to the **read** or **write** system call; otherwise, the count should be -1.

u.u_offset The file offset established by a previous **lseek** system call. Most character devices ignore this variable; but some, such as the **/dev/mem** pseudo device, use and maintain it. The value of **u.u_offset** is incremented after each character is transferred to or from **u.u_base**.

u.u_seg Determines the source and destination of the address space. Specifies what is in **u.u_base** as follows:

0 user instructions

1 user data

2 system data.

Field used by the *dd***select** entry point is:

u.u_procp A pointer to the process structure of a process that is waiting on a **select** system call.

Fields used by the multiplexed device drivers are:

u.u_dirp A pointer to the path name extension of a multiplexed device. For example, if an AIX process opens the multiplexed special device **/dev/hft**, the value of **u.u_dirp** is a NULL pointer. If an AIX process opens the multiplexed special device **/dev/hft/01**, the **u.u_dirp** field points to the string **01**. This is available to *dd***open**, and only when the **FDEVHANDLE** flag is set.

u.u_mpxchan A unique channel number associated with a multiplexed device. This field is updated by the driver's open entry point. Passed as a parameter to *dd***read**, *dd***write**, *dd***ioctl**, and *dd***select**.

C.4.2.2 Character Device Driver Entry Points

The following is a list of entry points from the kernel into a character device driver. Character device drivers have entry points that correspond with the **read**, **write**, **ioctl**, and **select** system calls.

---

ddread, **ddwrite**

```
ddread (dev, ext)
dev_t dev;
caddr_t;

ddwrite (dev, ext)
dev_t dev;
caddr_t;
```

The *dd***read** entry point is called by the kernel when a program issues a **read** system call on a file descriptor associated with a character device. The *dd***write** entry point is called by the kernel when a program issues a **write** system call on a file descriptor associated with a character device. The *dd***read** and *dd***write** entry points allow the device driver direct access to the user's address space.

Parameters:

dev Contains the device's major and minor number. Use the **major** and **minor** kernel subroutines.

ext The value of the **ext** parameter passed to the **readx** (extended read) or **writex** (extended write) system calls.

For most devices, the *dd***read** and *dd***write** entry points are synchronous. The waiting is accomplished by calling the **sleep** kernel subroutine and suspending the process in the device driver while it waits for the I/O to complete. Putting the process to sleep permits other processes to run.

Asynchronous *dd***read** and *dd***write** calls should be performed when the user opens the device with the **FNDELAY** flag. The device driver should consult the **u_fmode** field of the calling process to determine whether to perform asynchronous I/O for a device that can be opened for both asynchronous and synchronous I/O. Asynchronous *dd***write** calls can also be made by buffering the output data in kernel space and processing the buffers at interrupt time instead of process time. The type of device determines how the end-of-record and end-of-file is handled in *dd***read**.

When *dd***read** and *dd***write** entry points are provided for raw I/O to a block device, these entry points usually translate requests into block I/O requests. Some devices, such as block and block-oriented devices, require the user's buffer to be page-aligned.

The *dd***read** and *dd***write** entry points can indicate an error condition to the application program by storing a nonzero error code in the **u.u_error** field of the user block. If an error condition exists, the system call returns a value of -1 and the error code is available to the application in the **errno** external variable. The error code used should be one of the values defined in the **/usr/include/sys/errno.h** header file.

The *dd***read** and *dd***write** entry points can return partial completions by

updating **u.u_count** by the actual amount transferred.

---

ddioctl

```
dd ioctl (dev, cmd, arg, flag, ext)
dev_t dev;
int cmd;
caddr_t arg;
int flag;
caddr_t ext;
```

The *dd***ioctl** entry point of the specified device driver is called by the kernel when a program issues an **ioctl** system call. This entry point is a catch-all for special device-specific operations that do not fit into the **read/write** framework.

Parameters:

dev Contains the device's major and minor number. Use the **major** and **minor** kernel subroutines.

cmd The parameter from the system call that specifies the operation to be performed.

arg The parameter from the system call that specifies the address of a parameter block.

flag Specifies the flags passed on the open system call. See the **/usr/include/sys/file.h** header file for a complete definition of the bits in the **flag** parameter word. The **flag** parameter always returns ZERO.

ext The value of the **ext** parameter passed to the **ioctlx** (extended **ioctl**) system call.

Most **ioctl** operations depend on the specific device involved. However, all **ioctl** routines must respond to the following commands:

IOCTYPE Returns a character that indicates the device type.

IOCINFO Returns a structure that describes the device. This structure is defined as **devinfo** in the **/usr/include/sys/devinfo.h** header file. Only the first two fields of the data structure, **devtype** and **flags**, need to be set if the remaining fields do not apply to the device. For more information about this structure, see "devinfo" in topic 2.3.15.

All **ioctl** routines should be well-behaved. Well-behaved **ioctl** routines have their commands defined by the following macros:

_IOV(x,y) Uses the literal **arg** value; that is, the **arg** value is not used for copying data to and from the kernel.

_ION(x,y) There are no parameters for the **ioctl**.

_IOR(x,y,t) Copies data from kernel space to user space.

_IOW(x,y,t) Copies data from user space to kernel space.

_IOWR(x,y,t) Copies data from user space to kernel space and kernel space to user space.

In the above list,

x is the **ioctl** mask which is up to 8 bits long

y is the **ioctl** command which is up to 8 bits long

t is the length of the data area to be transferred to or from kernel space.

The following definitions for well-behaved **ioctl** routines are defined in the **/usr/include/sys/ioctl.h** file:

```
#define  TIOCGETD  _IOW(t,0,int)              /* get line discipline */
#define  TIOCSETD  _IOR(t,0,int)              /* set line discipline */
#define  TIOCHPCL  _ION(t,2)                  /* hang up on last close */
   .
   .

#define  TIOCSETP  _IOW(t,9, struct sgtty b)  /* set parms - gtty */
   .
   .

#define  SIOCGIFADDR _IOWR(i,13, struct ifreq) /* get ifnet address */
```

In well-behaved **ioctl** routines, the kernel instead of the device driver performs the copying of data from and to user space. Therefore, well-behaved **ioctl** routines use the **bcopy** subroutine instead of the **copyin** and **copyout** routines when copying multiple bytes of information to and from user space. These routines use simple assignment statements, rather than the **fubyte**, **fushort**, and **fuword** routines when copying single bytes, words, or long words of information to and from the kernel space.

The **ioctl** routines can also be ill-behaved. Ill-behaved **ioctl** routines have their commands defined as follows:

```
#define     SAMP_MASK       ('x'<<8)
#define     IOCTL1          (SAMP_MASK | 1 )
                    .
                    .
                    .
#define     IOCTLn          (SAMP_MASK | n )
```

Ill-behaved **ioctl** routines copy data directly to and from user space using the **copyin**, **copyout**, **fubyte**, **fuword** and **fushort** kernel subroutines.

The **ioctl** routines should be well-behaved, not ill-behaved, in order for the device driver to remain compatible with future releases of AIX.

The *dd***ioctl** entry point can indicate an error condition to the application program by storing a nonzero error code in the **u.u_error** field of the user block. When an error occurs, the system call returns a value of -1 and the error code is available to the application in the **errno** external variable. The error code used should be one of the values defined in the

/usr/include/sys/errno.h header file.

---

ddselect

```
#include <sys/select.h>
int ddselect (dev, seltype)
dev_t dev;
int seltype;
```

The *dd***select** entry point is used as an alternative to using signals for event notification. This entry point is called when the user program issues a **select** system call in order to determine whether or not an interesting event has occurred on the device.

Parameters:

dev Contains the device's major and minor number. Use the **major** and **minor** kernel subroutines.

seltype Specifies the type of selection operation with one of the following values:

FREAD	Read selection
FWRITE	Write selection
0	Exception selection.

The return values from *dd***select** are

0 Device is not ready. Collision occurred (that is, this is not the first process waiting on the event).

1 The selection criterion specified by **seltype** is true.

2 Device is not ready. This is the first process waiting on the event.

The *dd***read** and *dd***write** entry points and the exception-handling routines also require logic to support the **select** operation. Depending on how you write your device driver, your *dd***intr** entry point may need to include this logic as well. At each point where one of the selection criteria is true, the device driver checks for a process waiting for that selection and, if one exists, calls the **selwakeup** kernel subroutine to restart it. The collision flag and waiting process pointer that were saved are passed as parameters to **selwakeup**, and then they are reset.

The following example of a *dd***select** subroutine is adapted from the 3270 device driver:

```
int
tcaselect (dev, flag)
int     dev;
int     flag;
{
        register int selectDone;     /* select satisfied indicator */
        register int laNum;
        register linkAddr *laP;
        spl_t s;
```

```
        dev = minor(dev);

        laP = tca_data[dev].mlnk_ptrs[laNum];

        selectDone = 0;


        s=spl4(); /* mask intr. while checking read
                          and write device status */


        switch(flag){
        case FREAD:
                /* check to see if write to device
                         driver could be performed */
                if ((laP->io_flags & WDI_DAVAIL)){
                        selectDone = 1;
                } else {
                 /* check to see if another process is already sleeping
                 on read select of this device */
                        if(laP->dev_selr && selcoll(laP->dev_selr)){
                           /* set device select read collision flag */
                                laP->dev_flags |= RCOL;
                        } else {
                          /* save ptr to proc which MAY sleep on read */
                                laP->dev_selr = u.u_procp;
                                selectDone = 2;
                        }
                }
                break;

        case FWRITE:
                /* a write is always legal */
                selectDone = 1;
                break;
        case 0:
                /* check to see if exception occurred on interrupt */
                if(laP->io_flags & WDI_ALL_CHECK){
                        selectDone = 1;
                } else {
                  /* check to see if another process is already sleeping
                         on an exception select of this device */
                        if(laP->dev_sele && selcoll(laP->dev_selc)){
                     /* set device select exception collision flag */
                                laP->dev_flags |= ECOL;
                        } else {
                     /* save ptr to proc which MAY sleep on except */
                                laP->dev_sele = u.u_procp;
                                selectDone = 2;
                        }
                }
                break;
        }
        splx(s);    /* change mask back to what it was on entry */
        return(selectDone);
  }
```

Note in the above example that separate bits are turned on in **dev_flags**,

RCOL and **ECOL** when a collision occurs, enabling the routine issuing the **selwakeup** subroutine to know the type of collision that occurred. Also note that the **selcoll()** support routine is used to determine a collision. For more information, see "selwakeup" in topic C.6.2.2.

C.4.3 Block Device Drivers

Block device drivers provide an interface tailored for file system access in a random or pseudo random-access fashion. Examples of block devices include diskette, fixed disk, and magnetic tape.

When data is sent to or received from block devices, arbitrary user record lengths are mapped into fixed blocks. User data is buffered through kernel caching, thereby allowing for performance gains via asynchronous I/O. Because of the kernel's buffer cache, I/O for AIX applications is simpler for block rather than for character devices.

Subtopics
C.4.3.1 Block Device Data Structures
C.4.3.2 Block Device Driver Entry Points
C.4.3.3 Block Device Driver Data Flow
C.4.3.4 Block Device Kernel Subroutines

C.4.3.1 Block Device Data Structures

An area of memory is set aside within the kernel memory space for buffering data transfers between a program and the peripheral device. The kernel buffers are allocated in blocks of 4096 bytes. Each block becomes a member of one of two hashed chains that the device driver and the kernel maintain: the I/O chain or the free chain. The I/O chain contains those buffers for which I/O activity is in progress. The free chain contains the remaining buffers.

buf Structure: Each system buffer in the queue has a **buf** header structure that contains, among other information about the block, two sets of pointers to the next (**forw**) and previous (**back**) members in the list. The device driver maintains the available chain with the **av_forw** and **av_back** pointers. The kernel maintains the busy chain with the **b_forw** and **b_back** pointers.

The **buf** structure, which is defined in the **/usr/include/sys/buf.h** header file, includes the following fields:

Important Fields	Types	Description
b_flags	**long**	Flag bits.
b_forw	**struct buf***	Forward busy block pointer.
b_back	**struct buf***	Backward busy block pointer.
av_forw	**struct buf***	Forward pointer for a request queue.
av_back	**struct buf***	Backward pointer for a request queue.
b_dev	**dev_t**	Major and minor device number.
b_bcount	**bcount_t**	Byte count for the data transfer.
b_un.b_addr	**caddr_t**	Virtual memory address of the data buffer.
b_blkno	**daddr_t**	Block number on the device in units of **DEV_BSIZE**.
b_resid	**bcount_t**	Amount of data not transferred after error.
b_physaddr	**paddr_t**	Physical address, given to **dmasetup**.
b_error	**short**	Error number to give to user if **B_ERROR** is set.

In the buffer header, the **b_flags, b_forw, b_back, b_dev**, **b_count** and **b_un** fields are used by the system and may not be modified by the device driver. The **av_forw** and **av_back** fields are available for keeping a chain of such buffers by the kernel or by the device driver.

Note: The use of disk buffers by character device drivers is strongly discouraged. Instead, use the **malloc**, **kmemalloc**, or **palloc**

routines to allocate memory space.

iobuf Structure: Each block device driver has an **iobuf** structure which is used to maintain the I/O chain that is associated with the device driver. This structure contains two list heads: the **ib_forw/ib_back** list, which is a doubly-linked list pointing to all the buffers associated with that device driver; and the **ib_actf/ib_actl** list, which is used for the head and tail of the I/O chain. The driver only manipulates the **ib_actf/ib_actl** list and not the **ib_forw/ib_back** list.

The **ib_active** flag can be used to determine whether or not the device driver is currently processing a buffer or not. Typically, the ***dd*start** routine increments this flag when it initiates an I/O chain and then cancels it out when it processes the last of the buffers.

The device driver's **iobuf** structure also contains the following information:

The owner's major and minor numbe

An error record

Figure C-6 illustrates a block data structure. In the figure, the I/O chain is comprised of the **buf1** and **buf2** system buffers. The head and tail of the chain are **ib_actf/ib_actl**.

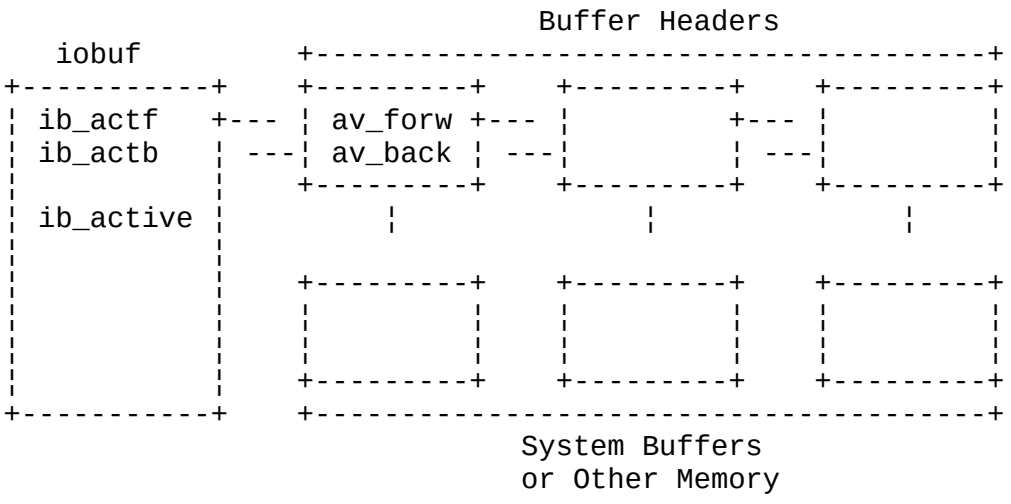


Figure C-6. Overview of Block Device Data Structures

C.4.3.2 Block Device Driver Entry Points

Besides the following entry points, block device drivers can have *dd***write** and *dd***read** routines that are used when the AIX application reads or writes to the raw block device, such as **rfd0** instead of **fd0**. When *dd***read** or *dd***write** are entered, the buffer cache is bypassed so that the driver can access the user's data just as it does for a character device driver. Most raw block devices are implemented by a call to the kernel subroutine **physio**, which formats a block request using the user's buffer, and invokes the driver's *dd***strategy** routine.

---

ddstrategy

*dd***strategy (bufp)**
struct buf *bufp;

The *dd***strategy** entry point is called internally by the AIX kernel to schedule I/O for swapping, file system I/O, and so forth.

Parameter:

bufp Points to a **buf** structure. The following information is important to *dd***strategy**:

- The type of transfer (read or write)
- Major and minor numbers
- The device block number (address on the device)
- The memory address to be used
- The byte count (number of bytes to be transferred)
- Flags describing completion (**B_ASYNC**).

A *dd***strategy** routine performs the following functions:

Checks the validity of the request (that is, whether the **b_blkno** value is in range). Bad requests are handled by setting an error code (usually E10) in the **b_error** field, setting **B_ERROR** in the **b_flags** field and calling the **iodone** routine.

Sets up the transfer of a multiple or single block sequenc

Orders buffers according to their physical position or location on th hardware device.

Schedules device I/O immediately by calling the *dd***start** routine

Block strategy routines are responsible for queueing buffers up fo I/O. The queue involves pointers and fields in which those pointers are put. The strategy routines are allowed to use the **avail** list pointers since a buffer that has pending I/O is not free to be re-allocated. In other words, the **avail** pointer fields are used to chain the buffer onto the free buffer list or the device I/O queue depending on the state of the buffer.

---

ddmbstrategy

ddmbstrategy (**flist**)
struct buf *flist;

Parameter:

flist A chain of buffers for submission to the strategy routine

The algorithm of this routine is to take each buffer on the list, validate its parameters and insert in onto the request queue. The request will be sorted onto the request queue, using an "elevator" algorithm, to minimize disk seek operations. The first request is then started if the device is idle.

```
  unit = -1;
  k = 0;
  /* if MBintr starts work on the queue while we are adding
   * items then it is inefficient but not an error
   * the critical section in enque is protected by an spl
   */
  while (flist) {
          bp = flist;
          dev = bp->b_dev;
          nu = drive(dev);
          flist = bp->av_forw;
          bp->av_forw = NULL;

          if ( (unit < 0) || (nu == unit) ) {
                    if ( ( nu = MBenque(bp) ) >= 0) {
                              unit = nu;
                              k++;
                    }
          }


          else {
                    printf("MBstrategy: unit=%d and unit=%d on queue ,

                                                  unit,nu);
                    bp->b_flags |= B_ERROR;
                    iodone(bp);
          }
  }
  if (unit >= 0 ) {
          s = spl6();

          fi = & MB_info ¦unit¦;
          dp = & fi->fi_tab;
          /* typically MBstart starts the I/O and b_active != 0
          * so we exit.  The alternative is that the sio fails
          * and b_active == 0 so we try next block
          */
          debug(FBADBG,("MBstrat: unit=%d active=%d actf=0x%x ,
                       unit, dp->b_active, dp->b_actf));
          while ( (dp->b_active == 0) && (dp->b_actf != NULL) )
                    MBstart(fi);
          splx (s);
  }
```

---

dddump (AIX PS/2)

```
dddump (dev, code, addr, count, offset)
dev_t dev;
int code;
paddr_t addr;
u_int count;
off_t offset;
```

For both the AIX PS/2 and the AIX/370 device drivers, the *dd***dump** entry point is called for kernel coredumps to write out a region of memory. When dumps are performed, no interrupts are delivered to the system. Dump routines must be able to operate by polling for I/O complete rather than relying on an interrupt.

Parameters:

dev Contains the device's major and minor number. Use the **major** and **minor** kernel subroutines.

code has one of the following values:

DUMP_INIT Initializes the dump (no data) and prepares the device to perform the dump. The **offset** parameter gives the total size.

DUMP_DATA Performs a dump of data pointed to by **addr**.

DUMP_END Specifies end of dump (no data). Handled like an end-of-file.

addr Specifies physical address of memory to dump.

count Specifies amount of memory (in bytes) to dump.

offset Specifies current byte offset into **dumpdev**.

---

dddump (AIX/370)

```
dddump(dumpdev, op, pageaddr, mp, len)
int op;
dev_t dumpdev;
int pageaddr;
register PTETYPE *mp;
int len;
```

Parameters:

dumpdev Contains the device's major and minor number.

op Specifies operation to be performed (B_READ or B_WRITE)

pageaddr Specifies the relative starting address of memory.

mp Specifies the pointer to a list of sequential page table entries.

len Specifies the length of transfer.

The *dd***dump** routine uses the **syncio** routine to perform I/O operations without accepting interrupts. The blocks of a dump device consist of a dump header describing each dump and a dump space map, and a large dump area. Each entry in the dump header describes a corresponding dump including time, size, the number of entries in the dumpmap, and the dumpmap. The dumpmap contains an entry describing each region in the dump. An entry is made for this dumps header, the blocks of the user structure and all of physical memory.

For both the AIX PS/2 and the AIX/370 device drivers, the default dump device is a dump minidisk set up by the **maint** utility during system installation. The user can override the default by inserting the following line into the **sysparms** stanza of the **/etc/system** file:

dumpdev = **special_file**

where **special_file** is the special file of the new dump device.

---

ddstart (AIX PS/2)

*dd***start**()

The *dd***start** entry point is called when the device is idle to determine if there is more work to do. The *dd***strategy** and *dd***intr** routines typically invoke *dd***start** to transfer data from system buffers to the hardware device.

If the task-time portion of the driver detects that the device idle, this routine may be called to start it. The *dd***start** entry point is also called by the interrupt handler to start the next request. Then *dd***start** checks whether the device driver is ready to accept another transfer request, and if so, it starts the request, usually by sending it a control word.

The *dd***start** entry point is not one that is placed in the device switch table. It is internal to block device drivers. Most block device drivers have *dd***start** routines, but this is not required.

---

ddstart (AIX/370)

ddstart (**deviceinfo**);
devinfo *device info*

Parameter:

deviceinfo Typically a pointer to the unit specific information structure.

The activity of the *dd***start** routine depends on the type of driver. A disk driver for example, must customize the CCWs for the validated request described by buffer header. This means translating the cylinder and other information into request appropriate for a typical disk driver. This routine is also responsible for building the IDAW list corresponding to the transfer.

A driver which performs I/O synchronous to the process (that is, the process sleeps, or relinquishes its ability to run in the driver until the I/O operation completes), might issue the request and set a flag stating that a process needed to be awakened (rescheduled for execution) when the operation completes.

A driver performing an operation to or from user virtual space must lock the pages involved in the transfer for the duration of the operation. The routines **physiolock** and **physiounlock** are provided for this purpose. An alternate interface is the **physio** interface which is typically used by block device drivers in supporting raw device interfaces. The driver may also take a timestamp before initiating the operation.

The *dd***start** routine for an AIX/370 driver must be able to handle instances when VM or another virtual machine may be using the physical device even though the AIX driver has not initiated any work.

Starting an operation on a busy device can fail if the channel cannot support multiple requests, or the channel, control unit or device are busy. The driver can detect this type of failure by noting the failure occurring when attempting to start the channel program. This is handled by performing a sense operation to clear the status and retrying the I/O operation up to a fixed number of tries.

AIX/370 I/O is started using the **sio** or **syncio** utility routines. The **syncio** routine acts as a synchronous operation and is only used during startup and for getting device status (sense)-- that is, during operations which complete in a fixed period of time).

C.4.3.3 Block Device Driver Data Flow

The following is a description of the data flow for buffers inside a tape device driver.

1. The ***dd*strategy** routine places the buffer on the back of the I/O queue.

2. If an I/O queue is currently not being processed then call the ***dd*start** routine.

3. The ***dd*start** routine turn on **ib_active** indicating that it is currently processing an I/O queue.

4. The ***dd*start** routine sends the data in the buffer to the device.

5. The device interrupts the driver when the buffer has been transferred.

6. The ***dd*intr** routine determines what buffer was processed, removes the buffer from the I/O queue, and frees the buffer by calling the **iodone** kernel subroutine.

7. The ***dd*intr** routine calls the ***dd*start** routine.

The following is an example of how to handle buffers in a tape driver:

```
struct iobuf tapetab;                /* iobuf for dd */
        .
        .
        .

/* the strategy routine */
tapestrategy ( new_buf )
        struct buf *new_buf;
{
        int x;

        .
        .
        .
        /* place the buffer on the back of the I/O queue */
        x = splblkio();
        if ( tapetab.ib_actf == NULL )
                tapetab.ib_actf = new_buf;
        else
                tapetab.ib_actl->av_forw = new_buf;

        tapetab.ib_actl = new_buf;


        if ( tapetab.ib_active == 0 )
                tapestart ();

        splx (x);
        .
        .
        .
}
```

```
/* the start routine */
tapestart ( )
{

        /* check to see if there is work to do */
        if ( tapetab.ib_actf == NULL )
        {
                tapetab.ib_active = 0;
                return;
        }



                  /* tell everyone that we are busy */
        tapetab.ib_active++;
                .
                .
                .
        /* Do controller specific commands to start I/O
         * on the buffer tapetab.ib_actf */
}


/* the interrupt handler */
tapeintr ( vec_num )
        int vec_num;
{
        struct buf *curbp;

        curbp = tapetab.ib_actf;
        if (error)
           curbp->b_flags| = B_ERROR
        else
           curbp->b_resid = bytesnotxfered;
        .
        .
        .
        iodone ( curbp );

        /* remove the spent buffer from the I/O queue
                and initiate the next transfer */
        if ( tapetab.ib_actf = spent_buf->av_forw )
        {
                tapestart ();
        }

}
```

The handling of buffers for disks and diskette drives is identical to that for tape devices, except that buffers are placed on the I/O queue as based on **b_cylin** in order to minimize disk head searching. Therefore, the **disksort** kernel subroutine is used in *dd***strategy** as follows:

```
        struct iobuf disktab;
        .
        .
        .
diskstrategy ( new_buf )
```

```
                struct buf *new_buf;
        {
                int x;

                .
                .
                .



                new_buf-> b_cylin = new_buf->b_blkno /
                            Num Sec Per Cyl;
                x = splblkio();
                disksort ( &disktab, new_buf );
                splx ( x );

                if ( tapetab.ib_active == 0 )
                        tapestart ();
                .
                .
                .

        }
```

---

ddenqueue

```
                ddenqueue (bp)
                struct buf *bp;
```

Parameter:

bp a buffer pointer representing an operation to be queued for asynchronous operations

The *dd***enqueue** routine is used to queue an operation onto an execution queue for a block driver. This routine is typically called from the **strategy** routine, the **multi-block strategy** routine, or may be a part of the **strategy** routine itself.

Its responsibility is to verify the request parameters with respect to the partition boundaries, requests reading off the end of a partition, verify transfer requests, and unit numbers.

This routine also calls the **disksort** routine to order the request amongst other driver requests to obtain the maximal throughput, by minimizing seek distance. The **disksort** routine uses an elevator algorithm in determining optimum ordering.

C.4.3.4 Block Device Kernel Subroutines

The following section details the routines necessary for disk buffer handling in terms of:

Block I/O buffer allocatio

Block I/O completio

Buffer cache management

AIX device drivers (even character device drivers) can get buffers from the system supply of available block I/O (disk) buffers.

Two kernel subroutines allow you to get and release buffers for use by your device driver. When disk buffers are used in AIX device drivers, the *dd***open** entry point typically calls **geteblk** to get the buffers, and *dd***close** calls **brelse** to return them.

The Block I/O Completion routines, **iowait** and **iodone**, are built on top of **sleep** and **wakeup** and are used to wait/post otherwise asynchronous block I/O.

The buffer cache management routines included **disksort** and **physio**. For more information, see "Virtual Address Space Management for DMA Devices" in topic C.6.1.6.

---

geteblk

struct buf *geteblk()

The **geteblk** kernel subroutine returns the address of a buffer header that is not in use. This routine always returns the address of a buffer header with an associated data area that is **BSIZE** bytes long and is page aligned. Note that **geteblk** does not allocate disk data buffers. If no free buffer headers are available, **geteblk** waits for one to become available. Therefore, you can call this routine only from the process level, not from interrupt context (*dd***intr**).

---

brelse

```
void brelse (bp)
struct buf *bp;
```

The **brelse** kernel subroutine frees a specific buffer. This kernel subroutine can be called from either interrupt context or process context.

Parameter:

bp Is a pointer to the buffer.

---

iowait

```
void iowait (bp)
struct buf *bp;
```

Parameters:

bp Is a **struct buf*** which addresses the buffer involved in the I/O operation.

The **iowait** kernel subroutine is called by the higher levels of the kernel I/O system in order to wait for the completion of an I/O operation specified by the buffer addressed by the parameter **bp**. This routine may call the **sleep** routine and therefore must not be called from an interrupt handler. The buffer status is stored in **u.u_error**.

---

iodone

```
void iodone (bp)
struct buf *bp;
```

The **iodone** kernel subroutine is called by the device driver when the block I/O transfer is complete. The **iodone** kernel subroutine marks the buffer pointed to by the **bp** parameter to indicate that the I/O has been completed. If the **B_ASYNC** bit of the buffer's **b_flags** field is set, indicating asynchronous I/O, the buffer is unlocked and returned to the free list. Otherwise, **iodone** wakes up any processes that are waiting for the buffer. This subroutine handles freeing of buffers and can be called from interrupt handlers. A driver should set the **b_resid** field appropriately if an error occurred (B_ERROR) in the **b_flags** field held before calling **iodone**(). If setting B_ERROR, a specific error code may be loaded into **b_error**. If **b_error** is left 0, EIO is assumed.

---

disksort

```
int disksort (disktab, bp)
struct iobuf *disktab;
struct buf *bp;
```

Parameters:

disktab Is the address of a **struct iobuf** which is declared within the driver to form the head of the I/O request queue.

bp Is a pointer to a buffer to add to the block I/O queue.

The **disksort** subroutine is called to add a block device I/O request to the queue of such requests for a particular device. It is normally called by the *dd***strategy** entry point. The **disktab** parameter is the head of the request queue, and the **bp** parameter addresses the buf structure containing the request. The queue of requests is sorted in ascending order by the **disksort()** routine, in a attempt to reduce disk head movement. The driver must initialize the **b_cylin** field of the **buf** header **bp** before calling

disksort since this is the field that **disksort** uses to order the queue.

---

physio

physio (**strategy, bp, dev, flag, checkcnt**)

Parameters:

strategy Is a pointer to the disk strategy routine for the block device.

bp Is a pointer to the buffer header describing the request to be filled.

dev Is an integer specifying the minor device number.

flag Specifies the I/O operation to be performed.

checkcnt Is a pointer to a routine used to enforce restrictions on transfer counts.

The **physio** kernel subroutine provides the raw I/O interface for block device drivers. It validates the request, builds a buffer header, locks the process in core, and calls the *dd***strategy** routine to place buffers on the I/O queue.

C.4.4 TTY Device Drivers

TTY device drivers are special character device drivers that interface to slow serial devices like terminals and printers. TTY device drivers use line discipline routines to erase a character or line, echo characters, and buffer input. The line discipline routines implement the **terminfo** interface for the hardware device.

Subtopics

C.4.4.1 TTY Device Driver Data Structures

There are several kernel structures that are used by TTY device drivers. These structures are:

- **tty** structures
- **clists**
- **cblocks**
- **ccblocks**
- **ttychars**
- **tty** maps.

C.4.4.2 tty Structures

A **tty** structure contains all the state information necessary to use an AIX TTY device for normal terminal I/O. Each TTY, or terminal device, has a **tty** structure associated with it. This **tty** structure contains the following:

Pointers to raw input and canonical queue

An output queue that holds all outbound character

Character control blocks for the input and output of character

Input and output state

The current column and ro

Link statistic

A pointer to the *dd***proc** routine

A structure of settable control character

Pointers to the process structure that are waiting to read or writ

A pointer to device-dependent information

The number of **tty** structures potentially allocated in the kernel at any one time is governed by the number of serial and parallel ports on the PS/2 as well as the **ntyunits**, **ptyunits**, and **x29units** system parameters, as set in the **/etc/master** or **/etc/system** file.

C.4.4.3 clist

A **clist** (character list) is a specialized data structure used to maintain a queue of characters. The **clist** structure is efficient and cheap storage, optimized for character-at-a-time applications, and most appropriately used for relatively slow devices such as terminals and printers.

For each **clist** that you use, you must define a queue header, which is a variable of type **struct clist**. This **clist** structure is defined in the **/usr/include/sys/tty.h** header file, and it contains the following members:

```
struct clist {
    int                 c_cc;       /* Character count */
    struct cblock       *c_cf;      /* Pointer to first block */
    struct cblock       *c_cl;      /* Pointer to last block */
};
```

You do not need to be concerned with maintaining the fields in the **clist** header; the character list kernel subroutines do that for you.

C.4.4.4 cblock

A **cblock** (**character block**) is a small buffer for contiguous characters. As defined by the **clist** structure, **clists** are ordered lists of **cblocks**. The **cblock** structure is defined in the **/usr/include/sys/tty.h** file as follows:

```
struct cblock {
   struct cblock    *c_next;         /* Pointer to the next block */
   char             c_first;         /* Offset to first character */
   char             c_list;          /* Offset to last character */
   char             c_data[CLSIZE];  /* Data */
};
```

A **cblock** does not need to be completely filled with characters. The fields **c_first** and **c_last** are zero-based offsets within the **c_data** array, which actually contains the data.

C.4.4.5 ccblocks

A **ccblock** (character control block) is a small buffer used between the interrupt handler and the user's application. A **ccblock** buffer is used during interrupt context to output and input a **cblock** of data. A **ccblock** has the following elements:

```
struct ccblock {
        caddr_t         c_ptr;      /* Buffer address */
        ushort          c_count;    /* Character count */
        short           c_size;     /* Buffer size */
};
```

C.4.4.6 ttychars

A **ttychars** structure (user settable control characters) contains control characters for a given TTY (terminal device). The elements of **ttychars** are used to cook inbound character streams from the raw queue to the canonical queue. Examples of control characters kept in this structure are:

Interrup

Qui

Erase last characte

Kill entire line

Some elements of **ttychars** are also used to control the hardware device. For example, **ttychars** contains the **tc_start** and **tc_stop** characters. The **tc_start** character is sent to inform the serial device to continue sending data. The **tc_stop** character is sent over the serial line when the line discipline routines determine that the internal buffers have reached a certain full threshold.

The **ttychars** structure is used primarily by the line discipline routines.

C.4.4.7 ttymaps

A **ttymap** structure is used in conjunction with **TCSMAP** and **TCGMAP** I/O control commands to set and obtain specific keyboard mapping characteristics.

C.4.4.8 TTY Device Driver Entry Points

TTY device drivers have the same entry points as character device drivers plus a *dd***proc** routine and a *dd***tty** entry point.

---

ddproc

```
*dd*proc (ptp, cmd)
struct tty *ptp;
int cmd
```

The *dd***proc** routine in the TTY device driver calls, and is called by, the line discipline routines. The line discipline routines can call *dd***proc** for any of the following reasons as specified by the **cmd** parameter:

T_OUTPUT Start output

T_TIME End of device timeout (restart output)

T_SUSPEND Suspend output (after receiving **t_stop**)

T_RESUME Resume output (after receiving **t_start**)

T_BLOCK Send a **t_stop** to throttle sender

T_UNBLOCK Send a **t_start** to resume sender

T_RFLUSH Flush **t_rbuf**

T_WFLUSH Flush **t_tbuf**

T_BREAK Set break condition on the line */

T_INPUT Start input (usually a no-op).

The *dd***proc** is not an entry point in the device switch table. It is internal to TTY device drivers and the address of *dd***proc** is stored in the device's **tty** structure during initialization of the **tty** structure.

The following is a sample *dd***proc** routine adapted from the serial device driver in the AIX kernel:

```
ddproc(tp, cmd)
register struct tty *tp;
{
        register int addr;
        switch (cmd) {

        case T_TIME:
                tp->t_state &= ~TIMEOUT;
                /* tell the transmitter to begin timeout */
                goto start;

        case T_WFLUSH:
                tp->t_tbuf.c_size -= tp->t_tbuf.c_count;
                tp->t_tbuf.c_count = 0;
```

```
        /* falls through */
case T_RESUME:
        tp->t_state &= ~TTSTOP;
        goto start;


case T_OUTPUT:
start:
        if (tp->t_state&(TIMEOUT|TTSTOP|BUSY))
                break;
        {
            register struct ccblock *tbuf;

            tbuf = &tp->t_tbuf;
            if ( tbuf->c_ptr == NULL || tbuf->c_count == 0 ) {
                if ( tbuf->c_ptr )
                        tbuf->c_ptr  -= tbuf->c_size
                                        - tbuf->c_count;
                if ( ! (CPRES &
                        (*linesw[tp->t_line]l_output)(tp)) )
                                break;
            }
            tp->t_state |= BUSY;
              /* output the character, pointer to by
                        *tbuf->c_ptr++, over the line */
                tbuf->c_count--;
        }
        break;


case T_SUSPEND:
        tp->t_state |= TTSTOP;
        break;


case T_BLOCK:
        tp->t_state &= ~TTXON;
        tp->t_state |= TBLOCK;
        if (tp->t_state&BUSY)
                tp->t_state |= TTXOFF;

        else
                /* send tp->t_stop over the line */
        break;

case T_RFLUSH:
        if (!(tp->t_state&TBLOCK))
                break;
case T_UNBLOCK:
        tp->t_state &= ~(TTXOFF|TBLOCK);
        if (tp->t_state&BUSY)
                tp->t_state |= TTXON;
        else
                /* send tp->t_start over the line */
        break;

case T_BREAK:
        /* send the break over the line */

        tp->t_state |= TIMEOUT;
```

```
    /* start a timeout by calling the timeout routine */
                    break;
            }
    }
```

---

ddtty

```
                    dd tty (dev)
                    dev_t dev;
```

The *dd***tty** entry point returns the address of the **tty** structure associated with a particular **tty** device or 0.

C.4.4.9 TTY Device Driver Data Flow

When a TTY device driver receives an inbound character, ***dd*intr** places the character into the read character control block and calls the input line discipline routine.
Note the following code:

```
ddintr ( vec_num )
 int vec_num;
{
 struct tty *tp;

  .
  .
  .

 /* data is lost if there is no room in the ccblock */
 if ( tp->t_rbut.c_ptr == NULL )
  return;

 *tp->t_rbuf.c_ptr = c;

 tp->t_rbuf.c_count--;
 (*linesw[tp->t_line].l_input)(tp, L_BUF);

  .
  .
  .


}
```

The line discipline routines maintain the raw input queue **clist** and increment **c_ptr**.

The driver's input routine is responsible for handling the translation of the break character. The following is an example of break handling:

```
            if ((c&0377) == 0)  {
                    if (flg&IGNBRK)
                            return;
                    if (flg&BRKINT) {
                            (*linesw[tp->t_line].l_input)
                                                    (tp, L_BREAK);
                            return;
                    }
            }
```

When an AIX application enters the ***dd*read** entry point, the TTY device driver calls the **l_read** line discipline routine. This routine either reads the data from the raw input queue or from the canonical queue, depending upon the input mode to which the user has set the terminal.

An outbound data stream enters ***dd*write** and the TTY device driver calls the **l_write** line discipline routine to place characters on an outbound **clist**. The **l_write** routine calls the ***dd*proc** routine to initiate outputting of the outbound data stream. After receiving an interrupt indicating I/O completion, ***dd*intr** invokes ***dd*proc** again to

continue with the next character in the data stream. Note the following example:

```
    ddintr ( vec_num )
    int vec_num;
    {
    struct tty *tp;

     .
     .
     .

    if ( tp->t_state & BUSY )
    {
                   tp->t_state &= ~BUSY;

     ddproc ( tp, T_OUTPUT );
    }

     .
     .
     .


   }
```

C.4.4.10 Line Discipline Routines

Two standard sets of line discipline routines are provided in the AIX kernel: one for terminals and one for line printers. Each of the routines is called through the line discipline switch table, **linesw**. The program segment which follows is a method of calling the functions which are defined in the **/usr/include/sys/conf.h** file:

```
(*linesw[tp->t_line].l_open)(dev,tp);
(*linesw[tp->t_line].l_close)(tp);
(*linesw[tp->t_line].l_input)(tp,code);
(*linesw[tp->t_line].l_read)(tp);
(*linesw[tp->t_line].l_write)(tp);
(*linesw[tp->t_line].l_output)(tp);
(*linesw[tp->t_line].l_ioctl)(tp,cmd,arg,mode);
```

If you are using the printer line disciplines, set the **t_line** parameter of the **tty** structure to 1 and the **t_dev** parameter to **DD_LP**. If you are using the **tty** line disciplines, set **t_line** to 0 and **t_dev** to **DD_TTY**.

The interface to either the **tty** or printer line disciplines is transparent to the TTY device driver.

---

l_open

```
l_open (dev, tp)
dev_t dev;
struct tty *tp;
```

The **l_open** line discipline routine is called on the first open of a a device by the *dd***open** entry point to perform the following functions:

Establish a process group for distribution of units and interrupt fro the TTY device

Initialize the **tty** structure

Prepare the device for I/O

Parameters:

dev Contains the device's major and minor number. Use the **major** and **minor** kernel subroutines.

tp Is the pointer to the device's **tty** structure.

---

l_close

```
l_close (tp)
struct tty *tp;
```

The **l_close** line discipline routine clears **tty** state information on final close of the device. This routine is called from the *dd***close** entry point.

Parameter:

tp Pointer to the device's **tty** structure.

---

l_input

```
l_input (tp, code)
struct tty *tp;
```

The **l_input** line discipline routine performs input character processing when called from the *dd***intr** entry point.

Parameter:

tp Pointer to the device's **tty** structure.

code **l_buf** to indicate chars received (in the **rbuf**) or **l_break** to indicate the receipt of a line break.

This routine places characters on the raw **tty** input queue, putting in delimiters and waking up any process waiting on input from this terminal. This routine also echoes characters, if required.

---

l_read

```
l_read (tp)
struct tty *tp;
```

The **l_read** line discipline routine performs specific input processing or waiting as called by the *dd***read** entry point. This routine transfers character from the raw queue, **t_rawq**, to the canonical queue, **t_canq**, counting delimiters as it goes.

Parameter:

tp Pointer to the device's **tty** structure.

---

l_write

```
l_write (tp)
struct tty *tp;
```

The **l_write** line discipline routine performs specific output processing as called from *dd***write**. This routine copies raw data into the output queue, **t_outq**, and calls the *dd***proc** entry point to initiate the transfer.

Parameter:

tp Pointer to the device's **tty** structure.

---

l_output

```
l_output (tp)
struct tty *tp;
```

The **l_output** line discipline routine places characters on the output character control block, **t_tbuf** and delays or expands tabs. This routine is called by the *dd***proc** entry point in the **T_OUTPUT** case when there are no characters left in **t_tbuf**.

The **l_output** routine returns a 0 if there are no more characters to output or **CPRES**.

Parameter:

tp Pointer to the device's **tty** structure.

---

l_ioctl

```
l_ioctl (tp, cmd, arg, flag)
struct tty *tp;
int cmd;
caddr_t arg;
int flag;
```

The **l_ioctl** line discipline routine performs specific **ioctl** handling such as initialization and cleanup. This routine is called in the **ttiocom** kernel subroutine and is generally not called from TTY device drivers.

Parameters:

tp Pointer to the device's **tty** structure.

cmd The parameter from the system call that specifies the operation to be performed.

arg The parameter from the system call that specifies the address of a parameter block.

flag Specifies the flags passed on the **open** system call. See the **/usr/include/sys/file.h** header file for a complete definition of the bits in the **flag** parameter word.

---

ttiocom

```
ttiocom (tp, cmd, arg, flag)
struct tty *tp;
int cmd;
caddr_t arg;
int flag;
```

The **l_ioctl** line discipline routine performs specific **ioctl** handling such as initialization and cleanup. The *dd***ioctl** entry point calls the **ttiocom** routine.

Parameters:

tp Pointer to the device's **tty** structure.

cmd The parameter from the system call that specifies the operation to be performed.

arg The parameter from the system call that specifies the address of a parameter block.

flag Specifies the flags passed on the **open** system call. See the **/usr/include/sys/file.h** header file for a complete definition of the bits in the **flag** parameter word.

Note: The following table shows a list of line discipline commands that are currently supported by **ttiocom**. Those commands followed by one asterisk (*) are for terminals only and those commands followed by two asterisks (**) are for printers only.

Figure C-7. Line discipline commands

IOCTYPE	IOCINFO	TIOCGETD	TIOCSETD
TIOCHPCL	TIOCGETP	TIOCSETP	TIOCSETN
TIOCOSM	TIOCNOSM	TIOCEXCL	TIOCNXCL
TIOCFLUSH	TIOCSETC	TIOCGETC	TIOCLBIS
TIOCLBIC	TIOCLSET	TIOCLGET	TIOCSLTC
TIOCGLTC	TIOCOUTQ	TIOCSTI	TIOCSTOP
TIOCSTART	TIOCGPAGE	TIOCSPAGE	TCGETA *
TCSETA *	TCSETAW *	TCSETAF *	TCSBRK *
TCXONC *	TCFLSH *	TCGLEN *	TCSLEN *
TCSMAP *	TCGMAP *	LPRGET **	LPRSET **
LPRGETV **	LPRSETV **	LPRVRMG **	LPRVRMS **
LPRUGES **	LPRUFLS **	LPRURES **	LPRGMOD **
LPRSMOD **	LPRGETA **	LPRSETA **	LPRGTOV **
LPRSTOV **	TIONREAD	TCKEP	LDIOC
LDOPEN	LDCLOSE	LDCHG	LDGETT
LDSETT	LDGETDT		

Note: I/O control commands for TTY devices are described in "termio" in topic 2.5.28.

---

ttinit

```
void ttinit (tp)
struct tty *tp;
```

Parameter:

tp points to the device's **tty** structure.

The **ttinit** kernel subroutine initializes a **tty** structure on the first open device. Generally called during the *dd***open** entry point.

---

ttyflush

```
void ttyflush (tp, cmd)
struct tty *tp;
int cmd;
```

Parameter:

tp points to the device's **tty** structure.

The **ttyflush** kernel subroutine flushes input and/or output queues and awakens any processing sleeping on input and/or output.

You can flush the input queue by setting the FREAD bit in **cmd** and you can flush the output queue by setting FWRITE. You can flush both queues by setting **cmd** equal to FWRITE/FREAD.

C.4.4.11 Installing Line Discipline Routines:

To install a set of line discipline routines into the AIX kernel, perform the following setp:

1. Create a stanza in **/etc/master**, like the following, that describes the line discipline:

```
lined:
        type = linedisc
        routines = open, close, read, write, ioctl, init, in, out
        softcft = true
        line = xx
        prefix = lined
```

   Note: **xx** in the above example MUST be a unique line discipline number.

2. Cause the line discipline to be linked into the AIX kernel, the next time newkernel executes, by placing a stanza similar to the following into **/etc/system**:

```
tline:
      driver = lined
      noshow = true
      noipl = true
      nospecial = true
```

3. Insert an initialization routine into the line discipline. The initialization routine will be called at boot time to install the rest of the entry points into the AIX kernel. Note the following sample initialization routine for a line discipline:

```
linedinit ( line )
int line;

    /* install the line discipline routine into the
    switch table */
   LDISC_INSTALL ( line,  linedinit, linedopen, linedclose,
                   linedread, linedwrite, linedioctl,
                   linedin, linedout, nulldev, "lined" );
```

4. Compile the line discipline using the following command:

```
cc -c -DKERNEL -D1386 lined.c
```

5. Archive the resulting .o file into the 386lib kernel archive by typing the following:

```
ar -rv /usr/sys/386/386lib.a lined.o
```

6. Rebuild the AIX kernel and Re-IPL

   After the kernel has been rebuilt, with the new line discipline routine added, AIX applications can access the line discipline by issuing the **TCSETA ioctl** command to the TTY device driver and setting the *c_line* parameter to the desired line discipline number.

C.4.4.12 TTY Device Driver Kernel Subroutines

The following kernel subroutines are provided for allocating and handling character lists. All of the routines mask the interrupts as needed so you can call them from either the process or interrupt level.

You can mix calls to the **getc**, **putc**, **getcb** and **putcb** routines. In this manner, you can insert characters in the buffer one by one, and remove them as a group. You can also insert characters as a group and remove them one by one.

The amount of system memory available for character queues is limited. The number of **cblock**s available to the system is defined by the **clists** keyword of the **sysparms** stanza in the **/etc/master** or **/etc/system** directory. All character device drivers must share this pool of memory. Therefore, you must limit the number of characters in your queue space to a few hundred. When the device is closed, the device driver should make certain that all of its character queues are flushed so that the character blocks are returned to the system.

---

getcf

```
struct cblock *getcf()
```

The **getcf** kernel subroutine gets a block from the free list and returns a pointer to it. If no blocks are available, **getcf** returns a NULL pointer. If you get buffer space with this routine, you must ensure that you free the space when you are through using it.

---

putcf

```
void putcf (p)
struct cblock *p;
```

The **putcf** kernel subroutine returns the block specified by the **p** parameter to the free block list.

---

getc

```
int getc (header)
struct clist *header;
```

The **getc** kernel subroutine returns the next character from the queue whose header is pointed to by the **header** parameter. If this character is the last in the buffer, the buffer is freed. If the buffer is empty, then **getc** returns -1.

---

putc

```
int putc (c, header)
char c;
struct clist *header;
```

The **putc** kernel subroutine puts the character **c** at the end of the queue whose header is pointed to by the **header** parameter. If the operation is successful, a value of 0 is returned. If the queue is full, **putc** automatically allocates new **cblocks** as needed. When no more blocks are available, **putc** returns -1.

---

getcb

```
struct cblock *getcb (header)
struct clist *header;
```

The **getcb** kernel subroutine returns the address of the buffer at the head the queue specified by the **header** parameter, or a NULL pointer if the queue is empty. The buffer is removed from the queue as well. If you get a buffer with this routine, you must ensure that you free the space when you are finished using it.

---

putcb

```
void putcb (p, header)
struct cblock *p;
struct clist *header;
```

The **putcb** kernel subroutine puts the buffer pointed to by **p** on the tail of the queue specified by **header**. Before calling this routine, you must load this buffer with characters and set **c_first** and **c_last**. The **p** parameter is the pointer returned by either the **getcf** or **getcb** routine.

---

getcbp

```
getcbp (p, cp, n)
struct clist *p;
char *cp;
int n;
```

The **getcbp** kernel subroutine copies **n** characters from the **clist p** to the buffer addressed by the **cp**. This routine frees a **cblock** from the queue if **n** is greater than the number of characters remaining in the **cblock**.

---

putcbp

```
int putcbp (p, cp, n)
struct clist *p;
```

```
char *cp;
int n;
```

The **putcbp** kernel subroutine copies characters from a buffer to the **clist** given as an argument. This routine returns the total number of bytes transferred to the **clist** and adds additional **cblocks** to the **clist** as needed.

---

puts

```
int puts (buffer, count, clist)
char *buffer;
int count;
struct clist *p;
```

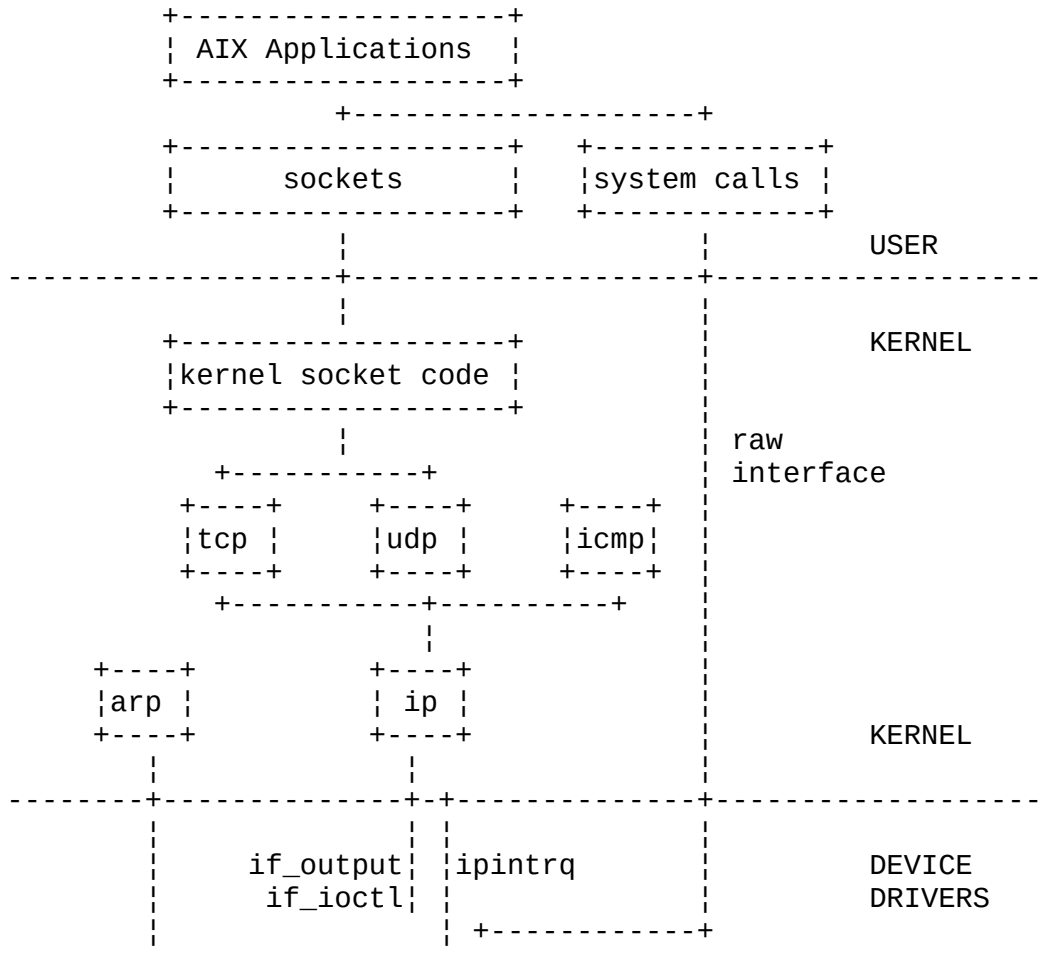
The **puts** kernel subroutine efficiently appends a buffer full of characters to a **clist**.

C.4.5 Network Device Drivers

Sockets provide an interface to network device drivers without the use of special files. Network device drivers are neither character nor block device drivers, although they can support a raw interface, similar to the way in which block device drivers can support a raw interface.

Provided the socket uses the internetwork address family, sockets call the kernel socket code which either uses the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP). TCP and UPD both call the Internet Protocol (IP) routines in the kernel. IP calls the **if_output** procedure handle of network device drivers to send outbound messages over the network or the **if_ioctl** procedure handle for special processing requests. If the network device driver needs to resolve the IP address into a physical address, the **if_output** procedure handle uses the Address Resolution Protocol (ARP) routines. The AIX kernel supports ARP routines for resolving IP addresses into Ethernet (802.3) and Token-Ring (802.5) physical addresses.

When the network device driver receives an inbound message, it checks to see if it is an IP or an ARP message. If the message has an IP address, the network device driver places the message onto ***ipintrq***, an input queue between all network device drivers and the IP routines, to send the data to the IP kernel routines. The IP routines route the data to either the TCP, UDP or the Internet Control Message Protocol (ICMP) routines. If the inbound message is an ARP message, the network device driver calls the ARP routines so that they may update their tables accordingly.



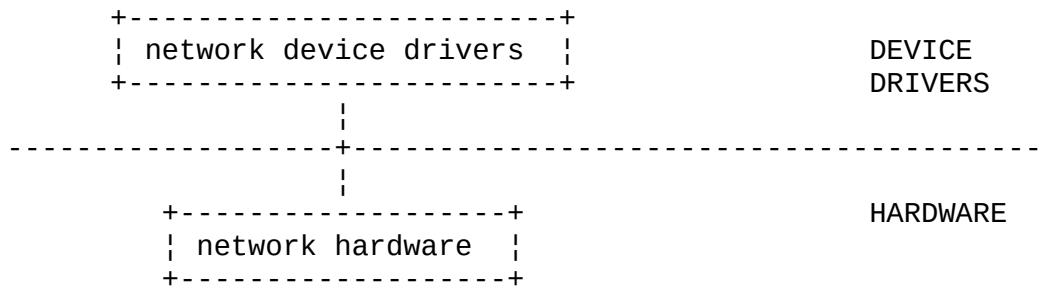


Figure C-8. Overview of AIX Network Device Drivers

Refer to Figure C-8 for an overview of the kernel components that send and receive data from network device drivers.

C.4.6 Network Device Driver Data Structures

Subtopics
C.4.6.1 mbufs
C.4.6.2 Network Interface Structure (ifnet)
C.4.6.3 IP Address Structures
C.4.6.4 ARP Structures

C.4.6.1 mbufs

When an AIX application sends a message to a foreign host, the message enters the network device driver formatted in an **mbuf** chain. The device driver formats inbound messages into **mbuf** chains and sends them to the kernel on the **ipintrq** input queue. Each **mbuf** in an **mbuf** chain contains a header with the following information:

A pointer to the next **mbuf** element in the chain (**m_next**)

A pointer to the next **mbuf** chain (**m_act**)

An offset to data **m_off**)

The type of mbuf **m_type**)

The amount of data in the **mbuf** chain, < MLEN (**m_len**)

A flag determining the type of externally associated buffer **mun_type**)

The free function to call if non **mbuf** cluster is attached (**mun_clfun**)

The argument passed to the free function if a non **mbuf** cluster is attached (**mun_clarg**).

Each **mbuf** chain designates one inbound or outbound message. Multiple **mbuf** chains may be linked together as shown in Figure C-9.

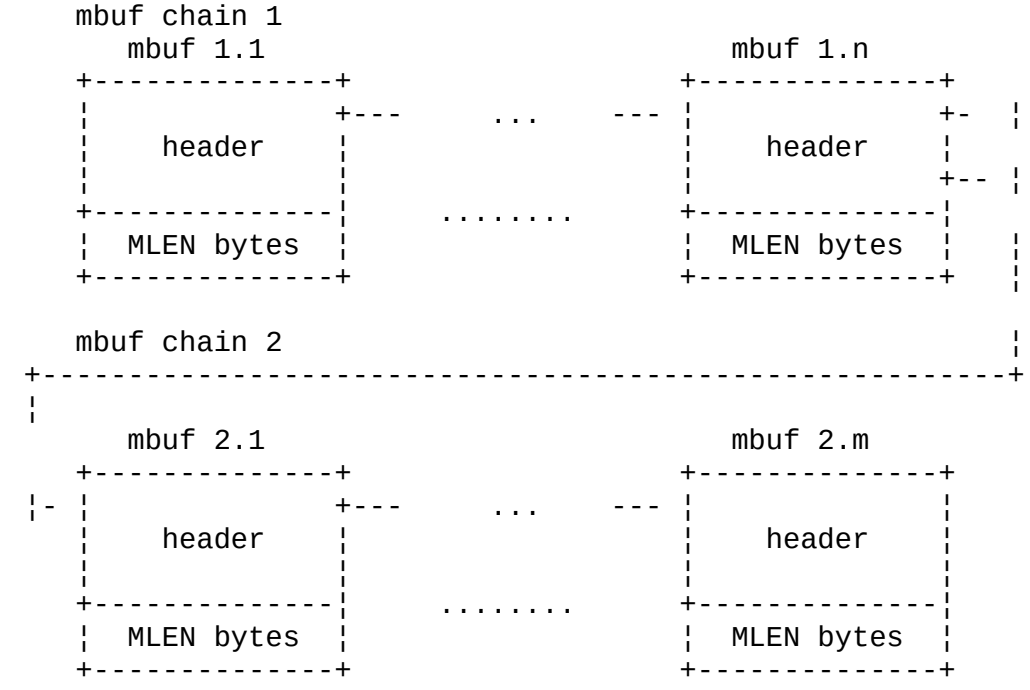


Figure C-9. Overview of **mbuf** Chains

When the kernel determines that an outbound message is greater than **mincluster** bytes in length, or when the network device driver determines that an inbound message is greater than **mincluster** bytes in length, the **mbuf** chain is made up of **mbuf** page clusters or a non-**mbuf** pool buffer.

When an element in an **mbuf** chain is an **mbuf** page cluster, the entire **mbuf** structure, including the data area, is considered the header to the **mbuf** cluster. The kernel uses a flag to the header field to determine whether this buffer contains an attached **mbuf** cluster or a non-**mbuf** pool buffer. Figure C-10 illustrates **mbuf** page clusters.

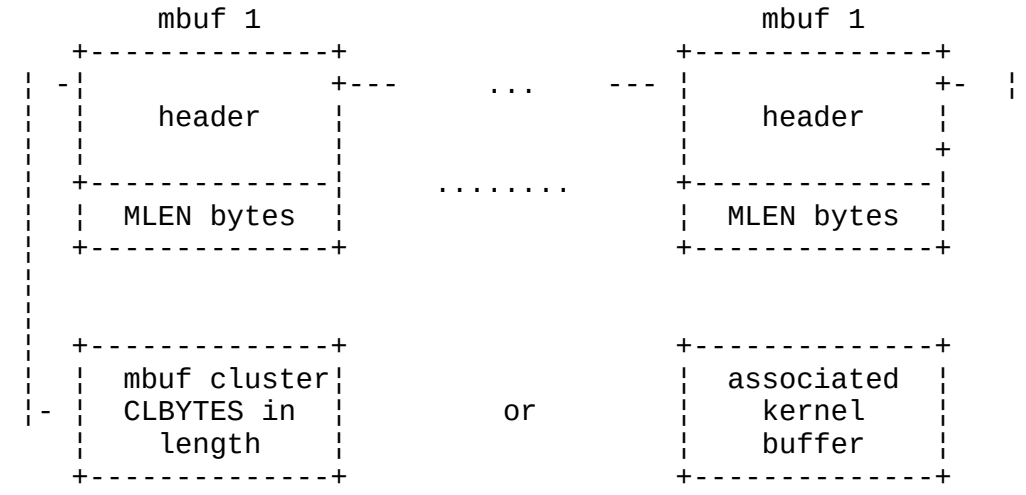


Figure C-10. Overview of **mbuf** Page Clusters

C.4.6.2 Network Interface Structure (ifnet)

Each port of each network adapter has a network interface structure (**ifnet**) associated with it. An **ifnet** structure contains general network interface information, output **mbuf** pointers, procedure handles, and link statistics. The **ddinit** routine supplies the general information concerning the network interface to **ifnet** and then attaches it to the AIX kernel via the **if_attach** kernel subroutine. The **ifnet** structure is defined in the **/usr/include/sys/if.h** file.

The following fields in the **ifnet** structure are of interest to network device drivers:

Interface name **if_name**). All ports supported by a device driver have the same interface name.

Sub unit. For each network adapter of a given interface name, this i a unique number (**if_unit**). For example, the AIX Token-Ring device driver supports up to two Token-Ring adapters. The adapter in the lowest slot number is assigned an **if_unit** of 0 and the next Token-Ring adapter is assigned an **if_unit** equal to 1.

Maximum transmission unit **if_mtu**)

Link status flags **if_flags**)

Address of the following procedure handles

if_init The initialization entry point (same as the **ddinit** routine)

if_output The routine called by the AIX kernel to handle all outbound messages

if_ioctl The routine called by the AIX kernel to handle special requests

if_reset A routine not supported by AIX

if_watchdog A watchdog timer

The number of seconds before the **if_watchdog** routine is called (**if_timer**).

The **if_addrlist** field points to a linked list of structures that describe the addresses of all foreign hosts with which the interface structure is currently communicating.

The network device driver maintains a queue of outbound **mbuf** chains in the **ifnet** structure. The following information on outbound **mbuf** chains is stored in the **ifnet** structure:

List header **if_snd.ifq_head**)

List trailer **if_snd.ifq_tail**)

Number of chains **if_snd.ifq_maxlen**)

Maximum length **if_snd.ifq_len**)

The **ifnet** structure also holds several link statistics which network drivers should maintain. They include:

if_ipackets Increment for every input packet received on a given network device.

if_ierrors Increment every time an input error occurs on a given network device.

if_opackets Increment for every output packet transmitted on a given network device.

if_oerrors Increment every time an output error occurs on a given network device.

if_collisions Increment every time a collision occurs on output (only relevant for CSMA interfaces such as Ethernet).

C.4.6.3 IP Address Structures

The **sockaddr** and **sockaddr_in** structures contain information that describes the address of the foreign host. The address of the destination is passed to **if_output** in the **sockaddr** structure, which is a generalized version of an address structure. The **sockaddr** structure is defined in the **/usr/include/sys/socket.h** file as follows:

```
struct sockaddr {
        u_short     sa_family;    /* address family */
        char        sa_data[14];  /* up to 14 bytes of direct address */
}
```

Since AIX currently only supports outbound messages that have the internet address AF_INET, **sockaddr** can be cast to **sockaddr_in**, the format of an internet address. The **sockaddr_in** structure is defined in the **/usr/include/sys/in.h** file as follows:

```
struct sockaddr_in {
    short    sin_family;
    u_short    sin_port;
    struct    in_addr sin_addr;    /* the actual internet address of
                                      foreign host */
    char    sin_zero[8];
};
```

Two additional IP address structures are **ifaddr** and **in_ifaddr** When the PS/2 first attempts to establish a connection to a foreign host, the AIX kernel calls **if_ioctl** with the **SIOCSIFADDR** command. An **ifaddr** structure is sent to the **if_ioctl** procedure handle. This structure contains the **sockaddr** structure of the foreign host address.
The **ifaddr** structure is defined in the **/usr/include/sys/if.h** file as follows:

```
struct ifaddr {
    struct    sockaddr ifa_addr;     /* address of interface */
    union {
        struct    sockaddr ifu_broadaddr;
        struct    sockaddr ifu_dstaddr;
    } ifa_ifu;
#define    ifa_broadaddr    ifa_ifu.ifu_broadaddr  /* broadcast address */
#define    ifa_dstaddr    ifa_ifu.ifu_dstaddr     /* other end
                                                     of p-to-p link */
    struct    ifnet *ifa_ifp;        /* back-pointer to interface */
    struct    ifaddr *ifa_next;      /* next address for interface */
};
```

The **if_addrlist** field in the **ifnet** structure contains a pointer to a linked list of **ifaddr** structures.

C.4.6.4 ARP Structures

Network device drivers and ARP routines share a structure called **arpcom**. This structure is defined in the **if_ieee802.h** file as follows:

```
  struct arpcom {
      struct ifnet ac_if;                /* network-visible interface */
      u_char ac_lanaddr[LAN_ADDR_SIZE];  /* LAN hardware address */
      struct in_addr ac_ipaddr;          /* copy of ip address */
  };
```

ac_if points to the **ifnet** structure for the particular adapter while **ac_lanaddr** and **ac_ipaddr** respectively contain the physical and IP address of a given network interface unit.

The relationship among **arpcom**, **ifnet**, and **ifaddr** is illustrated in Figure C-11.

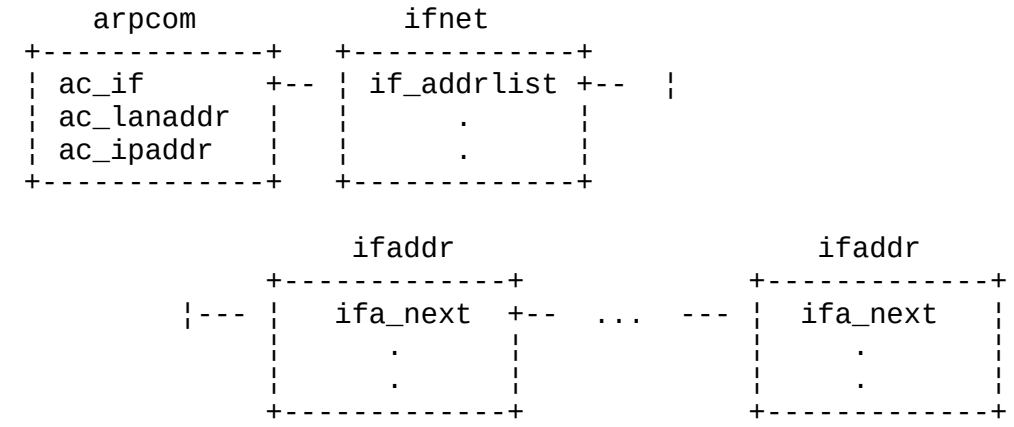


Figure C-11. Relationship among the **arpcom**, **ifnet** and **ifaddr** Structures

Two additional ARP structures are **arptab** and **ie5_arptab**. The **arptab** structure contains the internet to the physical address resolution table for Ethernet. The **ie5_arptab** structure contains the resolution table for Token-Ring. Both tables are defined in the **/usr/include/sys/if_ieee802.h** file. The format of the two resolution tables is as follows:

```
  struct arptab {
      struct in_addr at_iaddr;           /* internet address */
      u_char at_lanaddr[LAN_ADDR_SIZE];  /* LAN address */
      u_char at_timer;                   /* minutes since last reference */
      u_char at_flags;                   /* flags */
      struct mbuf *at_hold;              /* last packet until resolved/timeout
  };

  struct    ie5_arptab {
      struct    in_addr at_iaddr;        /* internet address */
      u_char    at_traddr[6];            /* token ring address */
      u_char    at_timer;                /* minutes since last reference */
```

```
    u_char    at_flags;              /* flags */
    u_short    at_rcf;               /* route control field        */
    u_short    at_seg[8];            /* routing info            */
    struct    mbuf *at_hold;         /* last packet until resolved/timeout
};
```

For each foreign host from which AIX is sending and receiving packets, there is an **arptab** table entry. When IP sends the **SIOCSIFADDR** command to the **if_ioctl** procedure handle, the network device driver calls the **arpwhohas** kernel subroutine. This routine sends an ARP message to all nodes in the network to determine the physical address of the internet address. When the node of interest responds, the ARP routines delete the old entry, if one exists, for the foreign host and create a new table entry.

When the IP output routine calls the **if_output** routine with a message having an IP header, **if_output** calls the **arpresolve** kernel subroutine. This subroutine searches the **arptab** table for the physical address corresponding with the internet address of the foreign host. If **arpresolve** does not find an entry in the **arptab** table for the foreign host, it saves the outbound message and then calls the **arpwhohas** routine.

If the host-to-host communication has been inactive for the number of minutes specified by **at_timer** minutes, the ARP routines delete the entry from the **arptab** table.

The values of **at_flags** are defined in the **/usr/include/sys/if_arp.h** file as follows:

ATF_INUSE Entry in use.

ATF_COM Completed entry (**enaddr** valid). This tells the ARP routines if the **arptab** table entry has a local interface, that is, whether an **ifaddr** structure associated with the foreign host has been placed into the linked list pointed to by **if_addrlist**.

ATF_PERM Permanent entry.

ATF_PUBL Publish entry (respond for other host).

ATF_USETRAILERS
Has requested trailers.

Another ARP structure is **lan_arp**. This structure specifies the broadcast frame that is used by the Ethernet device driver to resolve the physical address, and is defined in the **/usr/include/sys/if_ieee802.h** file.

C.4.7 Network Device Driver Procedure Handles

This section describes the network device driver procedure handles.

---

dd_output

```
dd_output (ifp, m0, dst)
struct ifnet *ifp;
struct mbuf *m0;
struct sockaddr *dst;
```

The **dd_output** routine takes work from the **ifnet** output queue and initiates the transmission of data over the network. The address of the **dd_output** routine is placed into the **if_output** field of the **ifnet** structure before the initial **if_attach** routine. The **dd_output** routine is then called indirectly through the **ifnet** structure by the kernel IP routines.

Parameters:

ifp Pointer to the request's network interface structure.

m0 Pointer to the **mbuf** chain containing the data to be sent over the network.

dst Pointer to the address structure of the destination.

Only three address families, as specified by **dst->sa_family**, are currently supported: **AF_INET**, **AF_UNSPEC**, and **AF_ARP**. The **AF_INET** messages come from the IP routines and the **AF_UNSPEC** and the **AF_ARP** messages are generated by the ARP routines.

AF_INET messages for Token-Ring and Ethernet must have their internet addresses translated by the **arpresolve** kernel subroutine. The **dd_output** routine creates a real header, usually containing the physical address of the host and destination and the packet type. This routine then allocates an **mbuf**, copies the real header into the **mbuf** and then places the **mbuf** on the front of **m0**.

The **arpresolve** routine also informs the network device driver whether or not to generate a trailer along with the outbound message. If a trailer is to be generated, then **dd_output** allocates an **mbuf** chain, inserts the trailer in the chain and then places the **mbuf** at the back of **m0**.

The **AF_UNSPEC** and **AF_ARP** messages are used by ARP to resolve internet addresses. These types of messages have their physical addresses already translated, and they contain the appropriate header and trailer. The **AF_UNSPEC** and **AF_ARP** messages are thus not manipulated by **dd_output**.

After the outbound message has been fully constructed, the **dd_output** routine places the message on the outbound queue pointed to by **ifp->if_snd**. Provided the message is queued successfully, this routine then calls the **start** routine to send the message over the link.

If an error occurs, the **if_output** routine can return one of the error codes for IPC/network devices as listed in the **/usr/include/sys/errno.h** file.

The following is the **dd_output** routine for the Token-Ring device driver:

```
tk_output(ifp, m0, dst)
struct ifnet *ifp;
struct mbuf *m0;
struct sockaddr *dst;
{
    register struct lan_llc_header *lh;
    register off;
    int type;
    struct lan_arp *ah;
    struct mbuf *m = m0;
    struct sockaddr_802_5 sa_tr;
    struct sockaddr_802_5 *sap = &sa_tr;
    struct in_addr idst;
    struct tk_softc *tk_softc;
    int error, usetrailers;
    short snap_type;
    int hdr_len, mac_len, llc_len;
    struct ie2_llc_hdr *llcp;
    struct ie5_mac_hdr *macp;
    spl_t s;

    /* Make sure that the net is up and running */
    if ((ifp->if_flags & (IFF_UP|IFF_RUNNING)) != (IFF_UP|IFF_RUNNING)) {
        error = ENETDOWN;
        goto bad;
    }


    /* Figure out the MAC destination address */
    switch (dst->sa_family) {
    case AF_INET:
        idst = ((struct sockaddr_in *) dst)->sin_addr;
        if (!ie5_arpresolve(&tk_softc->es_ac,m,&idst,sap,&usetrailers))
            return 0;
        off = ntohs((u_short)mtod(m, struct ip *)->ip_len) - m->m_len;
        if (usetrailers && off > 0 && (off & 0x1ff) == 0 &&
            m->m_off >= MMINOFF + 2 * sizeof (u_short)) {
            type = ETHERTYPE_TRAIL + (off>>9);
            m->m_off -= 2 * sizeof (u_short);
            m->m_len += 2 * sizeof (u_short);
            *mtod(m, u_short *) = htons((u_short)LANTYPE_IP);
            *(mtod(m, u_short *) + 1) = htons((u_short)m->m_len);
            goto gottrailertype;
        }
        off = 0;
        goto gottype;

    case AF_UNSPEC:
        debug(TOKENDBG, ("tk_ipc_output: AF_UNSPEC\n"));

        sap = (struct sockaddr *) dst;
        goto gottype;
    default:
        error = EAFNOSUPPORT;
    return
    }
```

```
gottrailertype:

    /*
     * Packet to be sent as trailer: move first packet
     * (control information) to end of chain.
     */
    while (m->m_next)
        m = m->m_next;
    m->m_next = m0;
    m = m0->m_next;
    m0->m_next = 0;
    m0 = m;


gottype:
    /*
     * Add local net header.
     *
     * Calculate the hdr length, ie5_mac_hdr + ie2_llc_hdr.
     */
    macp = &sap->sa_mac ;
    mac_len = mac_size( macp ) ;
    llc_len = sizeof( struct ie2_llc_hdr ) ;
    hdr_len = mac_len + llc_len ;

    /*
     * Find enough room for the headers.
     */
    if ( (m0->m_off > MMAXOFF) || (MMINOFF + hdr_len > m0->m_off) ) {
        m = m_get(M_DONTWAIT, MT_HEADER);
        if (m == 0) {
            error = ENOBUFS;
            goto bad ;
        }
        m->m_next = m0;
        m0 = m ;
        m0->m_off = MMINOFF;
        m0->m_len = hdr_len ;
    } else {
        m0->m_off -= hdr_len ;
        m0->m_len += hdr_len ;
    }


    /*
     * Fill in the mac header.
     */
    macp = mtod(m, struct ie5_mac_hdr *);
    bcopy( (caddr_t)&sap->sa_mac, (caddr_t)macp, mac_len ) ;

    /*
     * Fill in the llc header.
     */
    llcp = mac_to_llc( macp ) ;
    bcopy( (caddr_t)&sap->sa_llc, (caddr_t)llcp, llc_len ) ;

    s = splimp();
    if (IF_QFULL(&ifp->if_snd)) {
        IF_DROP(&ifp->if_snd);
        error = ENOBUFS;
```

```
            splx(s);
            goto qfull;
        }
        IF_ENQUEUE(&ifp->if_snd, m);
        tk_ipc_ostart(ifp);
        splx(s);
        return 0;
    qfull:
        m0 = m;
    bad:
        m_freem(m0);
        return error;
    }
```

---

dd_ostart

dd_ostart (ifp)
zstruct ifnet *ifp;

The **dd_ostart** routine is called when the device driver is idle to determine if there is more work to be done. The **dd_output** and **ddintr** routines typically call **dd_ostart** to transfer the next **mbuf** chain for a particular interface.

The **dd_ostart** routine checks to see that the device is ready, that is there is no transfer currently going on, and that there is an **mbuf** chain waiting to be sent in **ifp->if_snd**. If all of these conditions are true, **dd_ostart** removes the **mbuf** chain from the output queue, copies it to adapter memory, initiates the transfer over the network, frees the chain, and then increments **ifp->if_opackets**.

dd_ostart is not a procedure handle. It is internal to the network device driver.

Note the following example:

```
    dd_ostart(ifp)
    struct ifnet *ifp;
    {
        spl_t s;
        struct mbuf *m;

        /* check to make sure that we have a token ring board */

        /* set a global flag indicating that the board is busy */

        /* dequeue the mbuf chain */
        s = splimp();
        IF_DEQUEUE(&ifp->if_snd, m);
        splx(s);

        /* make sure that there is work to do */
        if (m == NULL) {
            /* set a global flag indicating that the board is not currently
```

```
    processing a buffer */
          return;
      }

      /* copy the mbuf chain to the adapter */

      m_freem(m);

      ++ifp->if_opackets;
  }
```

Subtopics
C.4.7.1 Input Processing

C.4.7.1 Input Processing

There is no input processing procedure handle. This is because inbound messages are handled by the **ddintr** routine. This routine merely queues inbound messages to the IP routines or hands them to the ARP routines for processing.

The interrupt handler should get the packet header and determine if the packet is either a **LANTYPE_IP** or a **LANTYPE_ARP**. If the message is a **LANTYPE_IP**, then the interrupt handler should perform actions similar to the following:

1. Increment **if_ipackets**

2. Copy the data frame into an **mbuf** chain

3. Call the **IF_ENQUEUE** routine to place the **mbuf** chain on the input queue

4. Notify the kernel that there is data to read by calling the **schnednetisr** kernel subroutine.

If the input is ARP protocol, simply call the **arpinput** routine to place the data onto the input queue.

The following input handler is based on the Token-Ring device driver:

```
dd_input()
 lbrc.
 struct ifnet *if;
 struct arpcom *arper;
 struct mbuf *m = NULL;
 int packet_type;
 spl_t s;
 char *all_data;
 char *addr_of_data,
 int data_len;
 extern struct mbuf *tkget();

 /* clear the interrupt */

 /*
  *  determine the length of the received packet.
  */

 /* let the adapter know that these receive buffers are free */
 tk_qasb(tk_board, RECEIVED_DATA, recv_buffer_addr);

 /* store the packet header.  Store the packet type in
  packet_type, the address of the data beyond
  the header in addr_of_data, the length
  of the data in data_len, set the interface
  pointer to if, and the arpcom struct to arper. */


 switch (packet_type){
 case LANTYPE_IP:

  m = tkget(addr_of_data, data_len, if );
  if (m == NULL)
```

```
   break;
  if (IF_QFULL(&ipintrq)) {
   IF_DROP(&ipintrq);
   m_freem(m);
   break;
  }
  IF_ENQUEUE(&ipintrq, m);
  schednetisr(NETISR_IP);
  break;
 case LANTYPE_ARP:

  ie5_arpinput(arper, all_data);
  break;
 }
}

struct mbuf *
tkget(addr, totlen, ifp)
register u_char *addr;
register int totlen;
struct ifnet *ifp;
{
 register int len;
 register struct mbuf *m;
 struct mbuf *top = NULL, **mp = &top;
 u_char *mcp;

 ++ifp->if_ipackets;

 while (totlen > 0) {
  MGET(m, M_DONTWAIT, MT_DATA);
  if (m == NULL)
   goto bad;
  len = totlen;
  if (ifp != NULL)
   len += sizeof(ifp);
    if (len >= mincluster) {
   MCLGET(m);
   if (m->m_len == CLBYTES)
    m->m_len = len = MIN(CLBYTES, len);
   else
    m->m_len = len = MIN(MLEN, len);
  }
  else {
   m->m_len = len = MIN(MLEN, len);
   m->m_off = MMINOFF;
  }

  mcp = mtod(m, u_char *);
  if (ifp != NULL) {
   * (mtod(m, struct ifnet **)) = ifp;
               mcp += sizeof(ifp);
               len -= sizeof(ifp);
               ifp = NULL;
          }
          bcopy(addr, mcp, len);
          addr += len;
          *mp = m;
          mp = &m->m_next;
          totlen -= len;
```

```
        }
        return top;
  bad:
        m_freem(top);
        return NULL;
  }
```

---

dd_ifioctl

```
dd_ifioctl (ifp, cmd, data)
struct ifnet *ifp;
int cmd;
caddr_t data;
```

The **dd_ifioctl** routine processes **ioctl** requests to the network device driver.

Parameters:

ifp Pointer to the request's network interface structure.

cmd Command to perform. Valid commands are:

SIOCSIFADDR
Set the **ifnet** address. For those protocols requiring ARP, the **SIOCSIFADDR** command determines the physical address of the foreign host by calling the **arpwhohas** kernel subroutine. This routine broadcasts a **LANTYPE_ARP** message to every node in the network to determine if the foreign host exists. If it does, the foreign host responds with an ARP message containing its physical address.

SIOCSIFFLAGS
Set the **ifnet** flags. As specified in the **/usr/include/sys/if.h** file, the values for flags can be as follows:

IFF_UP
Interface is up.

IFF_BROADCAST
Broadcast address valid.

IFF_DEBUG
Turn on debugging.

IFF_LOOPBACK
Is a loopback net. Not used by user-generated network device drivers.

IFF_POINTOPOINT
Interface is point-to-point link.

IFF_NOTRAILERS
Avoid use of trailers.

IFF_RUNNING
Resources allocated.

IFF_NOARP
No address resolution protocol.

IFF_PROMISC
Receive all packets.

IFF_ALLMULTI
Receive all multicast packets.

IFF_ETHERNET
Is an Ethernet device.

IFF_IEEE
IEEE 802 style Ethernet.

IFF_ALLCAST
Global broadcast for Token-Ring.

IFF_NOTCF
Not capable of forming a TCF network connection.

data Cast to **ifaddr** structure if issuing the **SIOCSIFADDR** command; not used by the **SIOCSIFFLAGS** command.

The following is a sample **dd_ifioctl** routine that is based on the Token-Ring device driver:

```
tk_ifioctl(ifp, cmd, data)
struct ifnet *ifp;
int cmd;
caddr_t data;
{
    register struct ifaddr *ifa = (struct ifaddr *) data;
    int error = 0;
    struct tk_softc softc;        /* IPC interface structure */

    switch (cmd) {
    case SIOCSIFADDR:
        ifp->if_flags |= IFF_UP;

        switch (ifa->ifa_addr.sa_family) {
        case AF_INET:
            ((struct arpcom *)ifp)->ac_ipaddr=IA_SIN(ifa)->sin_addr;
            arpwhohas((struct arpcom *)ifp, &IA_SIN(ifa)->sin_addr);
            break;
        }

       /* FALLS THROUGH */
    case SIOCSIFFLAGS:
        if (ifp->if_flags & IFF_UP) {
            if (ifp->if_flags & IFF_RUNNING)
                break;
            else {
                /* bring up the link */
            }ifp->if_flags |= IFF_RUNNING;
```

```
            } else {
                if (softc.es_if.if_flags & IFF_RUNNING) {
                    /* close down the link */
                } ifp-> if_flags & = ~IFF_RUNNING;
            }
            break;

        default:
            error = EINVAL;
            break;
        }
        return error;
    }
```

---

dd_watchdog

dd_watchdog (if_unit)
int if_unit;

The **dd_watchdog** routine can be used to periodically check status and states of the link. This routine is called after the time (in seconds) specified by **if_timer** has expired. The **dd_watchdog** routine does not affect performance. It is called internally, not as a result of socket calls from the user.

When you enter **dd_watchdog**, the value of **if_timer** is set to 0. You can reset the timer by setting **if_timer** to a nonzero value. After the value of **if_timer** expires, **dd_watchdog** is called again. The address of the **dd_watchdog** routine should be placed into the **if_watchdog** field of the **ifnet** structure before the **if_attach** is called in **ddinit**.

C.4.8 Kernel Subroutines for Network Device Drivers

Subtopics
C.4.8.1 mbuf Handling

C.4.8.1 mbuf Handling

There are two ways to manipulate **mbuf** chains: macros and routines. Macros are used for high performance and routines have more general purposes with slightly more overhead.

During any **mbuf** operation mask network interrupts by using the **splimp** routine.

MGET

```
MGET (m, canwait, type)
struct mbuf *m;
int canwait;
int type;
```

The **MGET** kernel subroutine gets a free **mbuf**.

Parameters:

canwait One of the following values:

M_DONTWAIT	Return immediately if there is no buffer available.
M_WAIT	Wait for a buffer to become available.

type One of the following values:

MT_FREE	Should be on free list
MT_DATA	Dynamic (data) allocation
MT_HEADER	Packet header
MT_SOCKET	Socket structure
MT_PCB	Protocol control block
MT_RTABLE	Routing tables
MT_HTABLE	IMP host tables
MT_ATABLE	Address resolution tables
MT_SONAME	Socket name
MT_ZOMBIE	Zombie process status
MT_SOOPTS	Socket options
MT_FTABLE	Fragment reassembly header
MT_RIGHTS	Access rights
MT_IFADDR	Interface address

On return, **m** is set a pointer to an **mbuf**; otherwise to 0 indicating an

error.

---

m_get

```
struct mbuf *
m_get (canwait, type)
int canwait, type;
```

The **m_get** kernel subroutine performs the same function as **MGET** except that it returns **m**. The **canwait** and **type** parameters are the same as in **MGET**.

---

m_getclr

```
struct mbuf *
m_getclr (canwait, type)
int canwait, type;
```

The **m_getclr** kernel subroutine performs the same function as **m_get** except that it also clears the buffer area after the **mbuf** is allocated. The **canwait** and **type** parameters are the same as in **MGET**.

MFREE

```
MFREE (m, n)
struct mbuf *m, *n;
```

The **MFREE** kernel subroutine frees an **mbuf** from the top of an **mbuf** chain.

Parameters:

m Pointer to the top of the **mbuf** chain.

n Pointer to the top of the **mbuf** chain after the **mbuf** has been freed.

---

m_free

```
struct mbuf *
m_free (m)
struct mbuf *m;
```

The **m_free** kernel subroutine performs the same function as **MFREE**, except that it returns **n**.

---

m_freem

```
void m_freem (m)
struct mbuf *m;
```

The **m_freem** kernel subroutine frees all **mbufs** in an **mbuf** chain.

MCLALLOC

```
MCLALLOC (m, canwait)
struct mbuf *m;
int canwait;
```

The **MCALLOC** kernel subroutine allocates an **mbuf** page cluster, containing CLBYTES of data space. Currently, the number of page clusters for AIX must be 1. Refer to "MGET" for the values of the **canwait** parameter.

Returns the pointer in **m**.

MCLFREE

```
MCLFREE (m)
struct mbuf *m;
```

The **MCLFREE** kernel subroutine frees **mbuf** page clusters allocated by **MCLALLOC**.

MCLGET

```
MCLGET (m)
struct mbuf *m;
```

The **MCLGET** kernel subroutine allocates an **mbuf** page cluster and then attaches the cluster onto the **mbuf** chain pointed to by **m**. Note that the **mlen** is set to CLBYTES on success and to 0 on failure (that is, if no **mbuf** page clusters are available).

---

mclgetx

```
mclgetx (fun, arg, addr, len, wait)
int (*fun)(), arg, len, wait;
caddr_t addr;
```

The **mclgetx** kernel subroutine allocates an **mbuf** header and then attaches the associated buffer to it. Saves the de-allocation function and argument to the **mbuf** header.

IF_ENQUEUE

```
IF_ENQUEUE (ifq, m)
struct ifnet *ifq;
struct mbuf *m;
```

The **IF_ENQUEUE** kernel subroutine places an **mbuf** chain onto the back of an **ifnet** queue. Refer to the **schednetisr** kernel subroutine for processing of inbound messages.

---

schnednetisr

```
#include <netisr.h>
schnednetisr (pup_level)
int pup_level;
```

The **schnednetisr** kernel subroutine generates a software interrupt to the 4.3BSD protocol handling routines inside the kernel to process an inbound network message that has been placed into an **mbuf** chain.

Parameter:

pup_level Indicates the protocol of the incoming message and may be one of the following values:

NETISR_RAW	unspecified protocol
NETISR_IP	internetwork: UDP, TCP
NETISR_IMP	arpanet imp (Internet Message Protocol) addresses
NETISR_NS	XEROX NS protocols.

IF_DEQUEUE

```
IF_DEQUEUE (ifq, m)
struct ifnet *ifq;
struct mbuf *m;
```

The **IF_DEQUEUE** kernel subroutine removes an **mbuf** chain from an **ifnet** queue. This subroutine must be used before **MFREE** or **m_freem**.

IF_PREPEND

```
IF_PREPEND (ifq, m)
struct ifnet *ifq;
struct mbuf *m;
```

The **IF_PREPEND** kernel subroutine adds an **mbuf** chain to the beginning of an **ifnet** queue.

IF_EMPTYQUEUE

```
IF_EMPTYQUEUE (ifq)
struct ifnet *ifq;
```

The **IF_EMPTYQUEUE** kernel subroutine returns TRUE if the **ifnet** queue is empty; otherwise, returns FALSE.

---

mtod

```
mtod (x, t)
struct mbuf *x;
t
```

The **mtod** kernel subroutine returns the pointer to the data area in an **mbuf** and casts the resulting pointer as **t**.

Parameters:

x Data address of an **mbuf**

t Cast of the returned address. For example, **char ***, or **struct ifnet ****

---

dtom

```
dtom (x)
struct mbuf *x;
```

The **dtom** kernel subroutine returns the address of the **mbuf** header.

Parameter:

x Data address of an **mbuf**.

H_HASCL

```
int H_HASCL (m)
struct mbuf *m;
```

The **H_HASCL** kernel subroutine returns a TRUE if **m** is an **mbuf** page cluster; otherwise, it returns a value of FALSE.

IF_QFULL

```
IF_QFULL (ifp)
struct ifnet *ifp;
```

The **IF_QFULL** kernel subroutine returns TRUE if the queue parameter **ifq_len** (the actual number of bytes in all the **mbuf** chains associated with a **ifnet** structure) is greater than **ifq_maxlen**; otherwise, it returns a value of FALSE.

IF_DROP

```
IF_DROP (ifp)
struct ifnet *ifp;
```

The **IF_DROP** kernel subroutine increments a counter of the number of packets dropped for a given **ifnet** structure.

---

m_copy

```
struct mbuf* m_copy (m, off, len)
struct mbuf *m;
int off;
long len;
```

The **m_copy** kernel subroutine makes a copy of an **mbuf** chain starting **off** bytes from the beginning, continuing for **len** bytes. If the value of **len** is **M_COPYALL**, copy to end of **mbuf** chain.

---

m_cat

```
m_cat (m, n)
struct mbuf *m, *n;
```

The **m_cat** kernel subroutine concatenates one **mbuf** chain, **n**, on the back of another, **m**.

---

m_pullup

```
struct mbuf *
m_pullup (n, len)
register struct mbuf *n;
int len;
```

The **m_pullup** kernel subroutine rearranges an **mbuf** chain so that **len** bytes are contiguous and in the data area of an **mbuf** (so that the **mtod** and **dtom** routines work for a structure of size **len**). Returns the resulting **mbuf** chain on success, frees it and returns NULL on failure. If there is room, the **m_pullup** routine adds up to **MPULL_EXTRA** bytes to the contiguous region in an attempt to avoid being called next time.

C.4.9 ARP Routines for Network Device Drivers

The following routines detail the interface to the AIX ARP routines for Ethernet and Token-Ring. If you want to support a different physical protocol that requires ARP, you would write routines that roughly correspond with those that follow:

---

arptimer

Ethernet:

```
arptimer ()
```

Token-Ring:

```
ie5_arptimer()
```

The **arptimer** kernel subroutine ages **arp_tab** and **ie5_arp_tab** entries once a minute. and checks to see if an entry in the **arptab** table has been accessed within a predefined threshold. **at_timer** counts the minutes since the last access.

This routine is invoked internally by the ARP routines and is not called by the network device driver.

---

arpwhohas

Ethernet:

```
arpwhohas (ac, addr)
register struct arpcom *ac;
struct in_addr *addr;
```

Token-Ring:

```
ie5_arpwhohas (ac, addr)
struct  arpcom *ac;
struct in_addr *addr;
```

The **arpwhohas** kernel subroutine broadcasts an ARP packet, asking who has **addr** on interface **ac**. This routine calls **dd_output** to output the ARP message.

---

arpresolve

Ethernet:

```
arpresolve (ac, m, destip, desten, usetrailers)
register struct arpcom *ac;
struct mbuf *m;
register struct in_addr *destip;
register u_char *desten;
int *usetrailers;
```

Token-Ring:

```
ie5_arpresolve (ac, m, destip,
daddr, usetrailers)
struct arpcom *ac;
struct mbuf *m;
struct in_addr *destip;
```

```
struct sockaddr_802_5 *daddr;
int *usetrailers;
```

The **arpresolve** kernel subroutine is called by the **dd_output** routine to resolve the internet address. The **arpresolve** routine returns a 1 if it is OK to send the packet because the ARP table already contains the physical address specified by the internet address; otherwise, a 0 is returned. If 0 is returned, then **arpresolve** stores the outbound **mbuf** pointer in the **at_hold** field of the **arptab** table entry so that the ARP routines can resend the buffer when the internet address is resolved.

If the **arpresolve** routine is successful for Token-Ring, **daddr** will contain the MAC and LLC headers required to send the packet. (The output routine must still deal with the variable size of the MAC.)

A return value of 1 for Ethernet indicates that the **desten** parameter has been filled in and the packet should be sent normally.

---

arpinput

Ethernet:

```
arpinput (ac, m)
struct arpcom *ac;
struct mbuf *m;
```

Token-Ring:

```
ie5_arpinput (ac, mac)
struct arpcom *ac;
struct ie5_mac_hdr *mac;
```

The **arpinput** kernel subroutine is called by the network interrupt handler when a network packet type **LANTYPE_ARP** is received. Common length and type checks are done and the protocol-specific routine, as defined in the ARP packet, is called. The **arpinput** routine calls the **in_arpinput** routine to process **LANTYPE_IP** or **IP_TRAILER** messages and takes care of formatting the inbound data stream into **mbufs**.

---

in_arpinput

Ethernet:

```
in_arpinput (ac, m)
register struct arpcom *ac;
struct mbuf *m;
```

Token-Ring:

```
in_ie5_arpinput (ac, mac, llc)
register struct arpcom *ac;
struct ie5_mac_hdr *mac;
struct ie2_llc_hdr *llc;
```

The **in_arpinput** kernel subroutine handles negotiations for use of trailer protocol. ARP replies for protocol type **ETHERTYPE_TRAIL** are sent along with IP replies. Trailers are also sent in response to IP replies. This allows either end to announce the desire to receive trailer packets. The **in_arpinput** routine also replies to requests for **ETHERTYPE_TRAIL** protocol

as well, but does not normally send requests. ARP for Internet protocols on Token-Ring or Ethernet. The algorithm used is that given in RFC 826. In addition, a sanity check is performed on the sender protocol address to catch impersonators.

This routine is internal to the ARP routines and is not called by the network device driver.

---

arptfree

Ethernet:

```
arptfree (at)
struct arptab *at;
```

Token-Ring:

```
ie5_arptfree (at)
struct ie5_arptab *at;
```

The **arptfree** kernel subroutine frees an **arptab** entry. Typically called by **arp_timer**, this routine is internal to the ARP routines and is not called by the network device driver.

---

arptnew

Ethernet:

```
struct arptab *
arptnew (addr)
struct in_addr *addr;
```

Token-Ring:

```
struct ie5_arptab *
ie5_arptnew (addr)
struct in_addr *addr;
```

The **arptnew** kernel subroutine enters a new address in the **arptab** table, eliminating the oldest entry from the bucket if there is no room. When required, the oldest entry is removed since no bucket can be completely filled with permanent entries (except from the **arpioctl** routine which tests whether another permanent entry will fit).

This routine is internal to the ARP routines and is not called by the network device driver.

C.5 ARTIC General Driver Support Routines

The following general comments apply to all of the ARTIC driver support routines described below.

The **unit** parameter refers to the index of the board as found by the POS registers; that is, board 0 is the ARTIC card in the lowest numbered slot in the AIX/PS2, board 1 is the next ARTIC found, and so on.

Addresses that refer to locations on the ARTIC card are linear addresses counting from 0 unless stated otherwise. The page number used to view the ARTIC memory is not preserved.

Task ID is the ID passed into the kernel by the ANOUNCETASK **ioctl** to the driver. By convention, the ID given to the ANOUNCETASK **ioctl** is the task ID in the task header. This is done by the **icaload** utility.

The default values for RCM maxtask, maxpri (maximum priority), maxqueue, and maxtimer are 16, 32, 16, 16 respectively. You can, however, change their default values in **/etc/system** by assigning each variable a new value which does not exceed 253. For example, to change the default values of maxtask and maxqueue, add the following lines in **/etc/system**:

```
maxtask = 30
maxqueue = 60
```

Next rebuild the kernel and reboot the machine for the changes to take effect.

Initially, all task interrupt handlers are set to an internal handler which will wake up a process that issued a command to the task with the ICACMD **ioctl**. If the task is not task 0, a diagnostic message gets logged to **/dev/osm** indicating that the interrupt occurred. If **icainteratch** has been called to install an interrupt handler for interrupts from a task, the wakeup and diagnostic messages do not happen. Also, **icainteratch** does not chain interrupts: it replaces the interrupt handler with the new one.

Refer to **/usr/include/sys/i386/ric.h** for using the ARTIC card under AIX/PS2; this file contains many useful definitions.

---

icastat

```
unsigned char icastat(unit, task)
int unit,
int unit,
```

Parameters:

unit ARTIC card to be queried

task Task to be queried

Return the primary status byte for the specified task on the specified board. Callable during **task** or interrupt time. This command saves and restores the value of the memory page register.

---

icacmd

```
int icacmd(flag, unit, task, cmd, arglen, arg)
unsigned flag, arglen;
int unit, task, cmd;
register caddr_t arg;
```

Send a command to the specified task and board.

This routine will spool the command if the output buffer or the task is busy. If the spooling flag is not set, this routine will return an error when the task is busy or the output buffer is needed and busy.

The spooling feature is not currently implemented, and the value of the flag argument is currently not looked at. The **icacmd** routine returns -1 if the task is not loaded and initialized, -2 if the task is busy, -3 if the command has an argument and the output buffer is busy, and -4 if the command argument is longer than the task's output buffer. The **icacmd** routine returns 0 upon success.

---

icarstr

```
void icarstr(unit, srcaddr, destptr, count)
int      unit, count;
unsigned long srcaddr;
caddr-t destptr;
```

Transfer count bytes from address *srcaddr* on the RIC card indicated by unit to the **destptr** location in kernel memory in the system unit. The *destptr* parameter is not checked for validity and will panic the kernel if it does not point to valid memory.

---

icawstr

```
void icawstr(unit, srcptr, destaddr, count)
int      unit, count;
unsigned long destaddr;
caddr-t srcptr;
```

Transfer count bytes from kernel memory in system unit to *destaddr* on the RIC card indicated by the unit argument. The *srcptr* parameter is not checked for validity and will panic the system if a bad pointer is passed in *srcptr*.

---

icaintratch

```
void icaintratch(flag, unit, task, funcp, arg)
int (*funcp) ();
```

Arrange for function pointed to by *funcp* to be called when the indicated task on the RIC card indicated by board posts an interrupt to the system unit with the INTPC **svc**. It will call the interrupt handler with the **board, task,** and **arg** (from the **arg** passed to this function) as arguments. All arguments of **funcp* are type **int**; that is, the calling sequence is (**funcp*) (*unit, task, arg*).

---

icafindtask

```
void icafindtask(unit, id)
int unit, id;
```

The **icafindtask** routine returns the task number of the first task found with a task ID matching ID on the RIC indicated by **unit**. If no task matching ID is found, -1 is returned.

The **icawaittask** routine

```
void icawaittask(unit, id)
int unit, id;
```

The **icawaittask** routine performs the same function as **icafindtask**, except that it sleeps until the desired task is loaded. Note that the caller sleeps indefinitely if the task is never loaded. If the task is loaded but not initialized it will wait 5 seconds for the task to initialize. If after 5 seconds the task is not initialized, **icawaittask** returns with the task number even though the task is not initialized. Since this call might call sleep, it cannot be used in an interrupt handler.

---

icagetbcb

```
void icagetbcb(unit, task, bcbaddrs)
int unit, task;
struct bcbptrs *bcbaddrs;

struct bcbptrs {
        long    statbuf;
        long    inbuf;
        long    outbuf;
};
```

The **icagetbcb** routine fills in the *bcbptrs* structure with values for the task and RIC card specified. The structure is filled with the linear value of the buffers computed from the page - offset values found in the task's BCB.

---

icarshort

```
short icarshort(unit, addr)
int unit;
long addr;
```

Return the short from address **addr** on the RIC indicated by **unit**.

---

icawshort

```
void icawshort(unit, addr, val)
int unit;
long addr;
short val;
```

Write the value *val* of type short at address *addr* on RIC indicated by **unit**.

---

icawchar

```
void icawchar(unit, addr, val)
int unit;
long addr;
char val;
```

Write the value *val* of type **char** at address *addr* on RIC indicated by *unit*.

C.6 Kernel Subroutines and Macros

This section details AIX kernel subroutines and macros in terms of:

Data transfer kernel routine

Process suspension and timin

Memory allocation and de-allocatio

Error handling and tracin

Masking interrupt

Determining the major and minor numbe

Determining the superuser

Subtopics

C.6.1 Data Transfer Kernel Routines

The following routines can be used for moving data for character I/O and between user and kernel space, moving user instructions between user and kernel space, manipulating kernel bulk data, transferring data to and from adapters, and virtual address space management for DMA devices.

Subtopics

C.6.1.1 Moving Data for Character I/O

Character device drivers can use the following kernel subroutines to transfer data into and out of the user space during **read** and **write** calls. These kernel subroutines use **u.u_base** as the address of the buffer in user space and **u.u_seg** to determine the type of data (kernel or user). These routines also automatically increment **u.u_base** and **u.u_offset** and decrement **u.u_count** by the number of bytes transferred.

The following kernel subroutines set the value of **u.u_error** to **EFAULT** if an invalid user space address is specified.

---

cpass

```
int cpass ()
```

The **cpass** kernel subroutine gets a character from the user buffer that is specified in a **write** system call. Upon successful completion, **cpass** returns the character. If the buffer is empty or if the user base address (**u.u_base**) points to a location outside user space, **cpass** returns a value of -1. The value of **u.u_error** is also set to **EFAULT**.

---

passc

```
int passc (c)
char c;
```

The **passc** kernel subroutine stores the character **c** in the user buffer that is specified in a **read** system call. Upon successful completion, **passc** returns a value of 0. If the buffer is full or if the user base address (**u.u_base**) points to a location outside user space, **cpass** returns a value of -1. If the address is invalid, **u.u_error** is also set to **EFAULT**.

---

iomove

```
void iomove (addr, count, flag)
char *addr;
int count, flag;
```

Moves a block of data between kernel space and user space.

Parameters:

addr Points to a buffer in kernel space

count Specifies the number of bytes to move.

flag Indicates the direction of the move:

B_READ Copies data from kernel space to user space
B_WRITE Copies from user space to kernel space.

If all or part of the user buffer is outside user space, **cpass** sets the value of **u.u_error** to **EFAULT**.

C.6.1.2 Moving Data between User and Kernel Space

The following kernel subroutines are at a lower level than the **cpass**, **passc**, and **iomove** routines and do not update the values in **u.u_error**, **u.u_count**, **u.u_offset**, or **u.u_base**. Use them to copy data between user and kernel space. If an error occurs during the transfer, the caller should set **u.u_error** to **EFAULT**.

---

subyte

```
int subyte (uaddr, c)
char *uaddr;
char c;
```

The **subyte** kernel subroutine stores the byte **c** at user data address **uaddr**. This routine returns a value of 0 upon successful completion, or a -1 if **uaddr** points outside user space.

---

suword

```
suword (uaddr, w)
int *addr;
int w;
```

The **suword** kernel subroutine stores the word **w** at user data address **uaddr**. This routine returns a value of 0 upon successful completion, or a -1 if **uaddr** points outside user space.

---

fubyte

```
int fubyte(uaddr)
int *uaddr;
```

Fetches the byte from user data address **uaddr**. **fubyte** returns the byte upon successful completion, or -1 if **uaddr** points outside user space.

---

fuword

```
int fuword (uaddr)
int *uaddr;
```

The **fuword** kernel subroutine fetches the word from the user data address **uaddr** and returns the word upon successful completion, or a -1 if **uaddr** points outside user space. Note that a legitimate value of -1 and the error indication are indistinguishable.

---

copyin

```
int copyin (uaddr, kaddr, count)
char *uaddr, *kaddr;
int count;
```

The **copyin** kernel subroutine copies **count** bytes from user data address **uaddr** to kernel data address **kaddr**. This routine returns a value of 0 upon successful completion, or a -1 if any or all of the **uaddr** buffer is outside user space.

---

copyout

```
int copyout (kaddr, uaddr, count)
char *kaddr, *uaddr;
int count;
```

The **copyout** kernel subroutine copies **count** bytes from kernel data address **kaddr** to user data address **uaddr**. This routine returns a value of 0 upon successful completion, or a -1 if any or all of the **uaddr** buffer is outside of user space.

---

fuscopy

```
int fuscopy (kaddr, uaddr, maxlen)
char *kaddr, *uaddr;
int maxlen;
```

The **fuscopy** kernel subroutine fetches a string from a user address **uaddr** and places it in **kaddr**. This routine copies data out of **uaddr** until it hits a NULL character or until **maxlen** bytes of data have been transferred; it then returns the total number of bytes transferred or a -1 if any or all of the **uaddr** buffer is outside of user space.

C.6.1.3 Moving User Instructions between User and Kernel Space

The following routines allow device drivers to access user instruction space. They are rarely used by an AIX device driver. User instruction kernel subroutines have identical parameters and meanings to their counterparts beginning on page C.6.1.2. For example, information on the **suibyte** routine appears under the **subyte** routine. Therefore, the kernel routine names are merely listed below:

suibyte
suiword
fuibyte
fuiword
copyiin
copyiout.

C.6.1.4 Manipulating Kernel Bulk Data

---

bcopy

```
bcopy (from, to, count)
caddr_t from;
caddr_t to;
int count;
```

The **bcopy** kernel subroutine causes a kernel FAST buffer move. This routine is only suited for intra-kernel movement. No address checking is performed on the buffers noted in **from** and **to**.

---

bzero

```
bzero (buf, count)
caddr_t buf;
int count;
```

The **bzero** kernel subroutine causes a kernel FAST buffer clear. This routine can only be used in kernel buffers. No address checking is performed on **buf**.

C.6.1.5 Transferring Data to and from an Adapter

Adapters can provide any of the four interfaces for transferring data to and from main memory:

Port I/

Memory-mapped I/

Direct memory access as a DMA slav

Direct memory access as a DMA master

Port I/O and memory-mappped I/O require the PS/2 processor to move the data. Either of the DMA mechanisms offloads the data movement from the PS/2 processor onto either the PS/2 system DMA controller (slave DMA) or a DMA controller on the adapter itself (master DMA).

Port I/O: Use the routines in this section to query hardware registers, set up DMA operations, clear interrupts, and move data to and from a device that does not support DMA.

---

ioin

```
ioin (port)
int port;
```

The **ioin** kernel subroutine returns a 16-bit word from the specified I/O **port**. The addresses of the I/O ports for a given adapter can be found in the documentation for that adapter. Note that they may be configured and that it may be necessary to examine the POS register data for the adapter to determine the exact addresses of the adapter's ports.

---

ioinb

```
ioinb (port)
int port;
```

The **ioinb** kernel subroutine returns an 8-bit byte from the specified I/O **port**.

---

ioout

```
ioout (port, val)
int port;
unsigned short val;
```

The **ioout** kernel subroutine outputs a (16-bit) word to the specified I/O **port**.

---

iooutb

```
iooutb (port, val)
int port;
unsigned char val;
```

The **iooutb** kernel subroutine outputs an 8-bit byte to the specified I/O **port**.

Memory-mapped I/O: Many PS/2 adapters provide memory-mapped I/O. This is a region of memory which is shared between the PS/2 processor and the adapter. It is accessed through normal memory-access instructions so that the kernel routines **bcopy()**, **bzero()**, and regular C assignments such as the following can be used to manipulate the data stored on the adapter:

```
*(vaddr+i) = 0x50;
```

Before assignments seen in the previous example can be made, the device driver must obtain a kernel virtual address which maps or gives access to the device's shared memory region. However, in order to obtain a kernel virtual address of the memory region, you must first know this address of the device memory. As with port address, the physical memory addresses for a given adapter can be found in the documentation for that adapter. Note that they too may be configured, and that it may be necessary to examine the POS register data for the adapter to determine the exact address of the adapter's memory. Once you know the physical address, the following macros can be used to provide the necessary kernel virtual addresses. Refer to the **/usr/include/sys/i386/mmu386.h** file for their definitions.

MAPIN

```
MAPIN (vaddr, paddr, bcnt )
caddr_t vaddr;
paddr_t paddr;
int bcnt;
```

The **MAPIN** macro contrives a (kernel) virtual address at which a physical address can be referenced. After the macro call, **vaddr** contains the virtual address.

Note: Set **vaddr** to NULL before calling **MAPIN**.

MAPIN_RO

```
MAPIN_RO (vaddr, paddr, bcnt)
caddr_t vaddr;
paddr_t paddr;
int bcnt;
```

The **MAPIN_RO** macro performs the same function as the **MAPIN** macro except that the AIX device driver is expected only to read the memory on the adapter (as opposed to both read and write).

Note: Set **vaddr** to NULL before calling **MAPIN_RO**.

Direct Memory Access as a DMA Slave: The PS/2 system provides a DMA controller which can be used to transfer data to and from device adapters that support system mode (slave) operations. The PS/2 micro-channel architecture allows for 16 different bus arbitration levels. Because multiple hardware devices may use the same arbitration level and since there are only 8 DMA channels to support the 16 arbitration levels, it is necessary for a device driver to allocate or reserve its arbitration level and a DMA channel before a DMA operation can be initiated.

Note the following rules when using DMA:

Always free the DMA channel after you perform your I/O functio

Do not unnecessarily tie up the DMA channel

DMA Resource Allocation Structure (dmaralloc): In order to perform DMA operations, a driver-supplied DMA resource allocation structure (**dmaralloc**) must be initialized by the AIX device driver and used as an argument to the DMA kernel subroutines. The **dmaralloc** structure contains the following fields:

dma_devicename Device name An ASCII string naming the device so that in the case of an error, the DMA routines can print out the name of the device affected.

dma_availfunc Function to call when DMA arbitration level and a suitable channel become available

dma_priority Device priority

dma_arblevel Device DMA arbitration level, usually from POS data.

The **dmaralloc** structure is defined in the **/usr/include/sys/i386/dmaralloc.h** file.

---

dmachanalloc

```
#include <i386/dmaralloc.h>
int dmachanalloc (ptr)
struct dmaralloc *ptr;
```

The **dmachanalloc** kernel subroutine allocates a DMA arbitration level and a suitable channel, locking out other devices from using the same arbitration level.

Returns the following values:

TRUE If the arbitration level specified by **ptr->dma_arb** was free and now has been assigned. The **dmachanalloc** routine also sets **ptr->dma_channel** to the channel that has been allocated and

sets the (**DMA_HAVECHAN**) flag in the **dma_flags** field.

FALSE If the arbitration level specified by **ptr->dma_arb** is busy. If a channel is not currently available and **ptr->dma_availfunc** is non-null, the arbitration structure is placed in a queue so that **ptr->dma_availfunc** is called when the arbitration level and a suitable channel become available. If **ptr->dma_availfunc** is null, no queueing takes place; it is assumed that the driver has made other arrangements for retrying the channel allocation.

Note: If the **dmachanalloc** routine returns FALSE and **ptr->dma_availfunc** is non-null, the DMA channel must still be allocated by a call to **dmachanalloc**. when the function pointed to by **ptr->dma_availfunc** is called. This, in fact, facilitates driver coding by allowing what is typically the driver start routine to serve as the **dma_availfunc**.

---

dmasetup

```
#include <i386/dmaralloc.h>

dmasetup (physaddr, func, count, dmarp, ioaddr, xfersize)
paddr_t physaddr;
long func;
unsigned short count;
struct dmaralloc *dmarp;
int ioaddr;
int xfersize;
```

Parameters:

physaddr The physical address of the data area. See the virtual address space management section for information on getting the physical address of a data area.

func One of the following values (from <**sys/buf.h**>):

B_READ Read data from the adapter into **physaddr**.

B_WRITE Write data to the adapter from **physaddr**.

count The number of bytes to transfer. If **xfersize** equals 2, **count** must be even.

dmarp Pointer to the DMA resource allocation structure that was used in the **dmachanalloc** routine.

ioaddr An I/O address to program into the DMA controller. This argument should be 0 for most devices, since it is the arbitration level which "connects" a DMA channel with a device.

xfersize Size of the DMA transfer cycle in bytes: 1 or 2 to indicate 8-bit or 16-bit transfers, respectively.

The **dmasetup** routine initiates the DMA transfer after the DMA channel has been allocated via the **dmachanalloc** routine. After the operation is complete, the hardware device typically interrupts the PS/2.

---

dmaresid

```
#include <i386/dmaralloc.h>
unsigned short dmaresid (ptr)
struct dmaralloc *ptr;
```

The **dmaresid** kernel subroutine returns the number of bytes that have not been transferred after a DMA transfer. If this function is invoked, it must be called before the channel is freed by the **dmachanfree** routine. The **dmaresid** routine is often used to set the **bp->b_resid** field of the **buf** header structure for block and raw block devices.

This routine is usually called by the **ddintr** entry point.

---

dmachanfree

```
#include <i386/dmaralloc.h>
dmachanfree (ptr)
struct dmaralloc *ptr;
```

The **dmachanfree** kernel subroutine de-allocates the DMA channel allocated by the **dmachanalloc** routine. **ptr->dma_arblevel** must contain the same arbitration level used in the **dmachanalloc** routine.

DMA Example: The following example shows you how to use the DMA kernel subroutines in an AIX device driver. This example assumes that the device will interrupt the device driver at the end of the DMA transfer.

```
int mtstart();

struct dmaralloc mtdma = {
        "Streaming tape",       /* device name */
        mtstart,                /* dma_availfunc */
        DMA_AVEPRI,             /* priority */
        0                       /* DMA arbitration level - to */
                                /* be filled in by mtinit */
};

mtstart()
{
        register struct buf *bp;

        kludge_flush();
        if ((bp = mttab.ib_actf) == NULL) {
                mttab.ib_active = 0;
                return;
        }
        mttab.ib_active = 1;
        if (! dmachanalloc(&mtdma))
                return;
```

```
        dmasetup(bp->b_flags & B_READ, bp->b_bcount,
                &mtdma, 0, sizeof(char));

        outb(MTCR, (ONLINE | INTENB | DACK | TC)); /* Tell adapter*/
                                                   /* to GO ! */
}


mtintr()
{
        register struct buf *bp;
        unsigned short resid = 0;

        kludge_flush();
        if ((inb(MTFBP1R) & INTTR) ==0
                return;                  /* not for us */

        if (mtdma.dma_flags & DMA_HAVECHAN) {
                resid = dmaresid(&mtdma);
                dmachanfree(&mtdma);
        }
        if (bp = mttab.ib_actf)
                bp->b_resid = resid;
        /* Clear the interrupt latch */
        outb(MTCR,(RIL | ONLINE));


        if (bp) {
                if (mtcurstate == MT_ERROR) {
                        bp->b_flags |= B_ERROR;

                if (mttab.ib_actf = bp->av_forw) {
                        iodone(bp);
                        mtstart();
                }
                else {
                        iodone(bp);
                        mttab.ib_active = 0;
                }
        }
}
mtinit(dev)
dev_t dev;
{
     int maj;
     int mtcardslot;

     maj = major(dev);
     if ((mtcardslot = devexist(MTCARDID, maj, 1, 0,)) != -1) {
               DEV_INSTALL(maj,mtinit,nulldev,mtopen,mtclose,
                        mtintr,B_TAPE|B_MINBLK);
               CDEV_INSTALL(maj,mtread,mtwrite,mtioctl,nulldev,notty);
               BDEV_INSTALL(maj,mtstrategy,nulldev,&mttab);
               intrattach(mtintr, 6, SPL_BLKIO);

                /* SET UP ARBITRATION LEVEL in dmaralloc struct */
                mtdma.dma_arblevel = devdata[mtcardslot].pd_pos3 & 0x0f;
      }
}
```

Direct Memory Access As a DMA Master: If the adapter card is equipped with a dedicated DMA controller, it can act as a DMA master and thus perform DMA operations without assistance from the AIX PS/2 DMA channels. For information on each adapter's use of its internal DMA controller, refer to the hardware reference manual.

C.6.1.6 Virtual Address Space Management for DMA Devices

Prior to transferring data to or from an adapter via DMA, the memory stored at a virtual address must be converted to a physical address and pinned, and the virtual address must be converted to a physical address. After the transfer of the data has completed, the memory must be unpinned. Note that pages mapped to be contiguous in virtual space are rarely contiguous in physical memory. Therefore, DMA transfers should never cross page boundaries except when it has been verified that those pages are physically contiguous. Virtual address space management takes three forms:

Read and write entry points of block device driver

DMA operations directly into/out of user spac

DMA operations directly into/out of kernel space

Read and Write Entry Points of Block Device Driver: Read and write entry points of block devices invoke the **physio** kernel subroutine. This routine faults in and pins the user buffer into memory, places the physical address of the buffer in **b_physaddr** field of the **buf** header provided by the driver, and then calls *dd***strategy**. **ddstart** then uses **b_physaddr** field as the **physaddr** argument to the **dmasetup** routine.

Device driver writers are encouraged to use the **physio** routine rather than perform virtual address space management themselves when performing DMA operations directly into and out of user space.

DMA operations Directly into and out of User Space: Prior to and after completing DMA operations into and out of user space, the device driver must perform the following steps on the user's virtual address:

1. Fault the address into memory using the **fubyte** routine.
2. If this is a read request, the device driver must issue the **subyte** kernel subroutine so that the copy-on-write gets handled correctly.
3. Call the **ubase2paddr** routine to translate **u.u_base** to a physical address.
4. Lock the page into memory by manually incrementing the page-frame reference count.
5. Perform the DMA operation.
6. When the interrupt is received on completion of the DMA, decrement the page-frame reference count.

Steps 1 to 5 must not be performed from an interrupt handler and only one page of memory should be pinned at a time.

The following example shows how to prepare the address for the DMA operation after the DMA channel has been successfully allocated:

```
int byte;


/* Make sure the page is in core */
```

```
if (( byte = fubyte (u.u_base)) < 0 )
{
        u.u_error = EFAULT;
        break;
}

if ( (flag == B_READ) && (subyte(u.u_base, byte)) < 0 )
{
        u.u_error = EFAULT;
        break;
}

/* translate the virtual address to a physical
        address */
paddr = ubase2paddr (flag);

/* lock the memory into core */
pfdat[ADDRTOPFN(paddr)]pf_lockcnt++;

/* perform the DMA operation */
```

In the interrupt handler, after the I/O is complete, unlock the above memory from core by issuing the following statement:

```
/* unlock the memory from core */
pfdat[ADDRTOPFN(paddr)]pf_lockcnt--;
```

---

ubase2paddr

```
paddr_t ubase2paddr (flag)
int flag;
```

The **ubase2paddr** kernel subroutine translates **u.u_base** into a physical address, checking that the memory in question exists and can be accessed as indicated by **flag**.

Parameter:

flag One of the following values:

B_READ Perform a read operation with the physical address

B_WRITE Perform a write operation with the physical address

If the access checks fail, the **ubase2paddr** routine sets **u.u_error** to **EFAULT** and returns **((paddr_t)-1)**.

DMA operations into/out of kernel space: All kernel space is automatically pinned into memory. Therefore, in order to perform DMA operations into and out of kernel memory, convert the kernel virtual address to a physical address before giving the address to the kernel DMA subroutines. The **kvtophys** kernel subroutine performs this conversion.

---

kvtophys

```
paddr_t kvtophys (cptr)
caddr_t cptr;
```

The **kvtophys** kernel subroutine converts a kernel virtual address to a 24-bit physical address. this routine is used primarily for converting addresses for DMA-type devices.

For example, suppose you want a block device driver to initiate its own I/O, perhaps to read a tape label or a disk volume table of contents (VTOC). The typical way to implement this is shown in the following example. Note the use of the **kvtophys** routine to convert the kernel virtual address to a physical address.

```
struct buf sampbuf;

        .
        .
        .

sampcmd ( dev, cmd, bufaddr, blkno, bcount )
        dev_t dev;
        u_char cmd;
        caddr_t bufaddr;
        daddr_t blkno;
        bcount_t bcount;
{

        /* wait for the driver buffer to become available */
        while ( sampbuf.b_flags & B_BUSY )
        {

                sampbuf.b_flags |= B_WANTED;
                sleep (( caddr_t) &sampbuf, PRIOBIO+1 );
        }


        sampbuf.b_flags = B_BUSY + B_PHYS;

        if ( cmd == WRITE )
                sampbuf.b_flags |= B_WRITE;

        if ( cmd == READ )
                sampbuf.b_flags |= B_READ;

        sampbuf.b_blkno = blkno;

        sampbuf.b_bcount = bcount;
        sampbuf.bun.b_addr = bufaddr;
        sampbuf.b_physaddr = kvtophys (bufaddr);
        sampbuf.b_dev = dev;

        sampstrategy ( &sampbuf );
        iowait ( &sampbuf );
```

```
        sampbuf.b_flags &= ¬B_BUSY;

        if ( sampbuf.b_b_flags & B_WANTED )
                wakeup ( (caddr_t) &sampbuf );

}
```

C.6.2 Process Suspension and Timing

The AIX operating system provides kernel subroutines to suspend and synchronize processes. These routines are described in this section.

Subtopics

C.6.2.1 sleep and wakeup

AIX device drivers typically suspend the calling process after issuing an I/O request by calling the **sleep** routine. When the interrupt from the hardware occurs, indicating that the I/O operation has completed, the **ddintr** routine calls the **wakeup** routine to start the process again.

The **sleep** and **wakeup** routines work anonymously. That is, the caller of the **wakeup** routine does not know what processes will be awakened and the recipient of the **wakeup** routine does not know what process was responsible.

The **sleep** and **wakeup** routines are primitive synchronization mechanisms. All processes sleeping on an event are awakened and the first process to run can not be determined. If a large number of processes are sleeping on a given event, it is desirable to implement some other sequencing mechanism so that processes are awakened one at a time and wasteful races are avoided.

---

sleep

```
int sleep (chan, pri)
caddr_t chan;
int pri;
```

The **sleep** kernel subroutine de-activates the calling process on channel **chan**. When the process activates again, it runs with the priority specified by **pri**. See the **/usr/include/sys/param.h** file for a list of valid priorities. The new priority is effective only while the device driver has control. Once control returns to the user program, the kernel controls the priorities.

The channel specified by the **chan** parameter is simply a value that identifies an event to wait for, or to sleep on. Addresses of global structures are often used but the **sleep** routine accepts any 32-bit value. The channel identifier must be unique system-wide.

The channel number is displayed in the **WCHAN** column of the **ps** command or in the **WCHAN** field of the **crash** subcommand **proc**. For more information on the **ps** and **crash** commands, see *AIX Operating System Commands Reference*.

Call **sleep** only from a specific process context and not from an interrupt handler.

---

tsleep

```
int tsleep (chan, pri, seconds)
caddr_t chan;
int pri, seconds;
```

The **tsleep** kernel subroutine is similar to the **sleep** routine except that it sleeps on a given channel for no more than the indicated number of seconds.

The **tsleep** routine returns the following values:

TS_OK Channel started normally by the **wakeup** routine.

TS_TIME Timeout occurred

TS_SIG Asynchronous signal occurred.

---

timesleep

```
int timesleep (chan, pri, ticks)
caddr_t chan;
short pri;
int ticks;
```

The **timesleep** kernel subroutine is similar to **tsleep** routine except that the time resolution is in units of 1/HZ seconds, where HZ is defined in the **/usr/include/sys/DOPTIONS.h** file.

C.6.2.2 Sleep and Signal Handling

A process may or may not be able to be interrupted while it is asleep. When the value of the **pri** parameter passed to the **sleep** routine is numerically less than or equal to the value of **PZERO**, which is defined in **/usr/include/sys/param.h**, the suspended process cannot be interrupted. In this case, signals that occur while the process is suspended do not interrupt it. If the value of the **pri** parameter is numerically greater than **PZERO**, the sleep can be interrupted and signals interrupt the suspended process.

Note: The **pri** parameter also sets the run priority of the process when it is activated.

When a signal interrupts a sleeping process, the process normally ends the sleep and restarts. The process does not return from the sleep, but uses the **longjmp** routine to return to the address saved in **u.u_qsav**. By default, **u.u_qsav** points to a routine that sets **u.u_error** (**errno**) to EINTR and returns the value -1 to the user process, indicating that the system call was interrupted by a signal. You can set the value of **u.u_qsav** to another address with the **setjmp**, **longjmp**, **_setjmp**, and **_longjmp** subroutines (see page 1.2.250).

If the process does not return from the sleep, the device driver cannot cleanup and free resources. Therefore, instead of letting the process return to the address saved in **u.u_qsav**, the process can also catch the signal in the device driver or you can cause the process to sleep at negative priority. To allow the process to catch the signal in the device driver, logically OR the value **PCATCH** with the **pri** parameter passed to the **sleep** routine. If the **sleep** routine returns because of a signal, it returns a nonzero value. Otherwise, it returns a value of 0.

This approach is demonstrated in the following example:

```
if ( sleep(chan, PZERO | PCATCH) != 0 )
{
/* ... Perform operations in response to the signal ... */
longjmp(&u.u_qsav);
}
```

In this case, a signal does not restart the process, but returns control to the next sequential instruction after the call to the **sleep** routine. This allows the kernel code to perform operations in response to the signal before returning normally.

The process can sleep at negative priority, but the device driver must guarantee that the process will be activated because the process cannot be stopped.

---

wakeup

```
void wakeup (chan)
caddr_t chan;
```

The **wakeup** kernel subroutine makes all processes that were suspended on

channel **chan** by the **sleep** kernel subroutine ready to execute. The processes do not actually begin to execute until the current process relinquishes control of the processor or returns to user mode. Because all processes that are waiting on the channel are restarted, a race condition occurs. Thus, after returning from the **sleep** kernel subroutine, each process should check to see whether it needs to be suspended again.

The **wakeup** kernel subroutine is usually called from the **ddintr** entry point and is called indirectly by the **iodone** routine for block devices.

---

wakeup_one

```
void wakeup_one (chan)
caddr_t chan;
```

The **wakeup_one** kernel subroutine is identical to the **wakeup** routine except that exactly one process waiting on the channel is made runnable. This routine is useful when all waiting processes are equal; that is, when it does not matter which process runs next. The advantage of using this routine is that only one process will run, yet there is no guarantee of fairness regarding the order in which processes are chosen to run.

---

selwakeup

```
void selwakeup (procptr, coll)
struct proc *procptr;
int coll;
```

The **selwakeup** kernel subroutine is used by the **ddintr** routine to awaken a process that has attempted to do a **select** system call when the awaited event finally occurs. The **ddselect** routine is responsible for storing the pointer to the process structure of the process, usually the first to enter **ddselect** for a given event, that is to be woken up when a given event occurs.

If the **coll** parameter is a nonzero value, the **selwakeup** routine knows that a collision was detected by the **ddselect** routine. Following collisions, the **selwakeup** routine causes all processes waiting on a given event to be activated. Only the process identified by **procptr** returns from the **select** system call. All other processes re-enter the **ddselect** routine to find out if a given event has occurred or if it has to wait for the next event.

The following example of the **selwakeup** routine in an interrupt handler matches the example provided in the description of the **ddselect** entry point:

```
tcaintr ( vec_num )
        int vec_num;
{

                        .
                        .
                        .
```

```
        /* we determine that there is data available */
        {

                        .
                        .
                        .

            /* wake those selecting on read */
            if(laP->dev_selr){
             selwakeup(laP->dev_selr, laP->dev_flags & RCOL);
             laP->dev_selr = NULL;
             laP->dev_flags &= ¬RCOL;
            }

                        .
                        .
                        .

        }


        /* we determine that there is an exception */
        {

                        .
                        .
                        .

            /* wake those selecting on exception */
            if(laP->dev_sele){
                selwakeup(laP->dev_sele, laP->dev_flags & ECOL);
                laP->dev_sele = NULL;
                laP->dev_flags &= ¬ECOL;
            }

                        .
                        .
                        .

        }

}
```

C.6.2.3 Kernel Timers

Kernel timers can be thought of as scheduled software interrupt mechanisms. Timers are useful for watchdog timers or fixed delays (breaks, speaker intervals, and so forth).

Kernel timers cause heavy cycle consumption if overused. One technique to avoid this problem is to have a general watchdog routine to reduce overhead. The watchdog routine should be entered periodically to check for events and update state variables.

Timeout routines are called at interrupt level and therefore must follow the conventions for interrupt handlers. For more information, see "General Considerations in AIX Device Drivers" in topic C.2.1.

Callout Structure: A **callout** structure is for routines arranging to be called by the clock interrupt with a specified argument after a specified amount of time.

The **callout** structure contains the following information:

Time of day for real interrupt

Time for virtual interrupt

Flags

C_CANCELABLE Will be cancelled; do not free

C_DONE Timeout has gone off

C_DELETED Timeout has been cancelled and should be ignored

Pointers to the next element in the **callout** list.

Device drivers do not manipulate **callout** structures, other than receiving a **callout** structure while initiating or terminating a cancellable timer.

C.6.2.4 Non-Cancellable Timers

---

timeout

```
void timeout (func, arg, ticks)
int (*func)();
int arg, ticks;
```

The **timeout** kernel subroutine schedules the function pointed to by the **func** parameter to be called with the parameter **arg** after the number of timer ticks specified by the **ticks** parameter. The **ticks** parameter is measured in units of IHZ, as defined in the **/usr/include/sys/param.h** file.

The kernel keeps track of pending **timeout** calls by keeping the information from calls made to **timeout** in a structure called the ***callout table***. This table can be examined with the **callout** subcommand of the AIX **crash** command.

---

delayticks

```
delayticks (ticks)
int ticks;
```

The **delayticks** kernel subroutine suspends the calling process for the number of timer ticks specified by the **ticks** parameter. The **ticks** parameter is measured in units of IHZ.

This routine is a wait loop and its use is highly discouraged due to the effect wait loops have on overall system performance.

C.6.2.5 Cancellable Timers

---

ctimeout

```
struct callout *ctimeout (func, arg, ticks)
int (*func)();
caddr_t arg;
int ticks;
```

The **ctimeout** kernel subroutine is similar to the **timeout** routine except that it must be cancelled via the **to_cancel** routine, whether or not the timer expires. A pointer to a **callout** structure is returned by the **ctimeout** routine. This pointer is used to cancel the timer by the **to_cancel** routine.

---

to_cancel

```
to_cancel (to_ptr)
struct callout *to_ptr;
```

The **to_cancel** kernel subroutine must be called after a **ctimeout** routine has been issued. The **to_cancel** routine returns a **timeout** event to the free pool. The **to_ptr** parameter is the pointer returned by the **ctimeout** routine.

Failure to issue a **to_cancel** routine after a **ctimeout** routine causes the **callout** structures to be lost. The **callout** structures allocated by the **ctimeout** routine have the **C_CANCELABLE** bit set in their **c_flag** field and are not automatically freed when the timer goes off.

C.6.2.6 Signals

The following signals allow AIX device drivers to directly notify user processes of certain events. Signals may be classified as follows:

Job Control	SIGHUP, SIGINTR, SIGQUIT, SIGCHLD, SIGTSTP, SIGSTOP, SIGCONT
Error	SIGSEGV, SIGTRAP, SIGILL, SIGFPE, SIGSYS
Communication	SIGALARM, SIGPIPE, SIGIO, SIGPOLL
Stop process	SIGKILL, SIGSTOP

Signals are posted in the process table of the recipient. If a process is sleeping at a positive priority, it is awakened. All processes check for signals at several key junctures:

Upon waking up from sleep at positive priority

Before returning to user mode

At certain clean places after negative priority sleeps

If pending signals are found, the process accepts one of the following consequences:

The process ignores the signal and nothing happens

The process catches the signal, pushing a new call frame onto th stack.

The process is stopped, possibly causing a user core dump of th process and causing the **exit** system call to be called.

The following kernel subroutines may be called inside of device drivers to cause and process signals.

---

psignal

```
void psignal (p, sig)
struct proc *p;
u_int sig;
```

The **psignal** kernel subroutine sends a signal to a process.

Parameters:

p Points to the process table entry for the receiving process.

sig Specifies the signal to send.

To get the value for the **p** parameter, save the **u.u_procp** file, which contains a pointer to the process table entry for the process that made the system call. Remember that the user structure and hence **u.u_propc** may not be accessed at interrupt time.

For a list of the valid signals and more information about how signals work, see "sigaction, sigvec, signal" in topic 1.2.263.

---

gsignal

```
void gsignal (groupid, sig)
pid_t groupid;
u_int sig;
```

The **gsignal** kernel subroutine sends a signal to all the processes with **groupid** as their process group. To get the value for **groupid**, save **u.u_procp->p_pgrp**. The **gsignal** routine is typically used by **tty**-type devices to send the SIGHUP, SIGINTR, and SIGQUIT signals.

ISSIG

```
void ISSIG (pp)
struct proc *pp;
```

The **ISSIG** kernel subroutine returns the signal number if the current process has a signal to process; otherwise, it returns 0.

The **ISSIG** routine is typically used in the following context:

```
pp = u.u_procp;
if (pp->p_cursig || ISSIG(pp) || pp->p_sig_arg[MIG_SIGARG])
psig();
```

---

psig

```
void psig ()
```

The **psig** kernel subroutine performs the action specified by the current signal. This routine may or may not return a value, depending on the action taken.

C.6.3 Memory Allocation and Deallocation

If AIX device drivers expect to buffer more than a few characters of data, they should use the dynamic storage facilities provided by the kernel.

---

kmemalloc

```
#include <vmalloc.h>
caddr_t kmemalloc (nbytes, flags)
int nbytes;
u_int flags;
```

The **kmemalloc** kernel subroutine allocates bytes of memory as defined by **flags**.

Parameters:

nbytes Number of bytes of memory

flags One of the following values:

Alignment flags

MA_DBLWD Align on a double-word boundary

MA_PAGE Align on a page boundary

MA_DBLWDNOPAGE
Align on a double-word that is not on a page boundary.

Time of duration

MA_LONGTERM Memory is not freed for more than 10 minutes, or at all

MA_MIDTERM Memory is not freed within 10 minutes

MA_SHORTTERM Memory is freed within seconds.

Other flags

MA_OK2SLEEP Indicates that the process should sleep, waiting for memory to become available before returning.

The **kmemalloc** routine can be called from both processes and interrupt routines, although the **MA_OK2SLEEP** flag must not be on when allocating memory from an interrupt routine.

If the memory cannot be allocated, **kmemalloc** returns a NULL pointer. Callers must always check the return value of **kmemalloc** for NULL, even if the **MA_OK2SLEEP** flag is passed.

---

malloc

```
caddr_t malloc (nbytes)
```

u_int nbytes;

The **malloc** kernel subroutine allocates kernel memory and is based on the **kmemalloc** routine. When using the **malloc** routine, memory is allocated on a double word boundary and is freed within 10 minutes. If memory is not available, the process will be suspended until sufficient memory can be allocated. This routine can only be called from process level.

Note that **malloc** is not the same routine as those found in the **lib.c** file, despite the identical name.

---

palloc

caddr_t palloc (size, align)

The **palloc** kernel subroutine returns a pointer to an area of length aligned on an address boundary of 2 raised to a specified power. This routine is based on the **kmemalloc** routine.

Parameters:

size Area of length on the address boundary

align Power used to raise specified byte area.

For example, **palloc (1024,11)** returns a pointer to a 1024-byte area that is aligned on a 2048-byte boundary (p = 2048). The **palloc** routine returns a NULL pointer if the requested block of memory cannot be allocated. This routine can only be called from process level.

When using the **palloc** routine, memory is aligned on a doubleword but not a page boundary and is freed within 10 minutes; if memory is not available, the process is suspended until sufficient memory can be allocated.

Note that this is not the same routine as that found in the **lib.c** file, despite the identical name.

---

mfree

void mfree (ap)
caddr_t ap;

The **mfree** kernel subroutine frees the memory pointed to by **ap**. This routine can be called from process level or interrupt level and must not be called with anything that was not the return value from the **kmemalloc**, **malloc**, or **palloc** routines.

C.6.4 Error Handling and Tracing

Kernel errors can be handled in one of four ways:

Logging messages to the consol

Logging messages to the error logge

Reflecting errors back to the use

Taking the system down

The kernel tracing mechanism can be used for tracking conditions that lead up to an error.

Subtopics
C.6.4.1 Logging Messages to the Console
C.6.4.2 Logging Messages to the Error Logger
C.6.4.3 Reflecting Errors to the User
C.6.4.4 Taking the System Down
C.6.4.5 Trace Logging

C.6.4.1 Logging Messages to the Console

---

printf

void printf (format, value1, value2, ...)

The **printf** kernel subroutine writes a formatted character string to the **/dev/osm** file and then writes the string to the console.

Warning: Most of the system's operation is suspended while **printf** is writing to the console so use this routine only for important messages.

The **printf** kernel subroutine resembles the **printf** subroutine described in Chapter 2 Volume I, but the two should not be confused. The latter subroutine is part of the **libc** library, which is used by application programs. The kernel subroutine, on the other hand, is built into the kernel and is accessible only within the kernel and device drivers. In addition, the **printf** kernel subroutine recognizes only the %s, %d, %o, %x, %c, and %p conversion specifications as well as the 'l' (long modifier) and simple field width specifications, for example:

```
"%8lx"
```

Field width, precision, and other modifiers are not recognized. Noting these important differences, see "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf, wsprintf" in topic 1.2.208 for detailed information about the **format** parameter.

It is common to use **printf** routines in device drivers as debugging aids in the early stages of driver development; however, most **printf** routines should be removed as the driver reaches production level. Errors should be reported through the **errsave** kernel subroutine.

---

ncprintf

void ncprintf (format, value1,
value2, ...)

The **ncprintf** (non-console **printf**) kernel subroutine writes a formatted character string to the **/dev/osm** file, but not to the console. This routine is used to output diagnostic messages, such as the addresses found and locations in the device driver that were entered and other information that the user does not care about but may be useful in problem determination.

---

putchar

void putchar (c, touser)
int c;
int touser

The **putchar** kernel subroutine prints one character to the system console or user's terminal. This routine does a **busy wait** rather than depending on interrupts.

If the **touser** parameter has a nonzero value, the character is outputted to the user's terminal instead of to the system console.

C.6.4.2 Logging Messages to the Error Logger

The error log data structure is used in combination with the **errsave** kernel subroutine to enter device driver errors into the AIX error logger. A record sent to the error logger consists of a header and data. The format of the data is specified in the **/etc/errfmt** file.

Assume that the error record for a device driver is declared as follows:

```
#include <erec.h>
struct e_errsave err;
```

Then the following fields of the driver error structure must be filled out before calling **errsave** with an error record:

err.es_csmt
Contains the error class, subclass, mask, and type. Values for class, subclass, and type are provided in the **/usr/include/sys/erec.h** file. The mask value is used to further subdivide an error subclass.

err.es_stlen
Set to 12.

err.es_drvstat.e_intlvl
Device interrupt level

err.es_drvstat.e_dmalvl
DMA arbitration level

err.es_cardid
The 2-byte adapter ID

err.e_diag
9 bytes of device-specific information that is formatted by the **/etc/errfmt** file.

---

errsave

```
#include <erec.h>
errsave (info, len)
struct e_errsave *info;
int len;
```

The **errsave** kernel subroutine is the special purpose error logging routine for standard device errors.

As long as the **errdemon** is running, all errors entered by **errsave** are placed into non-volatile RAM so that errors may be recovered after reboot from a kernel **panic** routine.

C.6.4.3 Reflecting Errors to the User

There are four ways to reflect errors to the user:

Setting the **u.u_error** field

Indicating that an exception occurred during a **select** system call

Signalling a proces

Posting an error in a buffer

Setting u.u_error Field: Setting the **u.u_error** field results in a negative value being returned to the user's program from a system call. The **u.u_error** field may only be manipulated in the context of the requesting process and not from an interrupt handler. In addition, only certain system calls: **ddopen**, **read**, **write**, and **ioctl** may set **u.u_error**.

Note: Do not set the **u.u_error** field from a **ddclose** entry point.

Exception during select System Call: Character device drivers allow the user to indicate that an exception occurred a **select** system call. For more information on the **ddselect** entry point, see "Character Device Driver Entry Points" in topic C.4.2.2. If an error occurs prior to the exception, the **ddselect** entry point notifies the kernel that an exception has occurred and the process **select** system call is satisfied immediately.

If an error occurs after the kernel suspends a process waiting on the **select** system call, the following events occur:

1. The device driver detects that an error occurred, usually in the **ddintr** routine.

2. The device driver issues the **selwakeup** kernel subroutine, thereby satisfying the process **select** system call.

Signalling a Process: The interrupt handler usually signals a process provided that you have previously stored the process ID of the recipient during process context (during a read or write, for example).

Posting an Error in a Buffer: If, while performing I/O on a disk buffer, the device driver detects an error, it can OR the B_ERROR bit in the buffer's **b_flags** parameter and optionally set **b_error** with a value from the **usr/include/sys/errno.h** file. There is one important side effect to setting errors in buffer headers: once the B_ERROR bit is set in **b_flags**, the buffer is no longer used in the buffer cache.

Buffers can only be reflected back to the user if the B_ASYNC bit is not turned on in **b_flags** (that is, the read or write is synchronous). If the driver does not set **b_error**, the kernel sets **u.u_error** field to EIO.

Posting errors in disk buffer headers is usually accomplished in the **ddintr** routine.

C.6.4.4 Taking the System Down

---

panic

```
panic (s)
char *s;
```

The **panic** kernel subroutine is called when a catastrophic error occurs and the system can no longer continue to operate. It performs the following actions:

Uses the **printf** routine to write the character string pointed to by the **s** parameter to the console, preceded by the word **panic**
Does a system dum
Records the first 14 characters of the **s** string in non-volatile random access memory (NVRAM)
Reboots the system

---

icpanic

```
icpanic (name, number, file, panic_flags)
char *name;
int number;
char *file;
int panic_flags;
```

The **icpanic** kernel subroutine is called when an inconsistency panic occurs and is typically used in file system code. It prints a standard inconsistency message that indicates a bug in the kernel. This routine should not be used to report resource exhaustion.

Parameters:

name Name of the function where the failure occurred

number Line number in the source file or a unique error code

file Name of the source file where the failure occurred

panic_flags If nonzero, the error is outputted as being non-fatal and the **panic** subroutine is not called. This flag is usually 0.

C.6.4.5 Trace Logging

---

trsave

```
void trsave (traceid, cnt, buf)
unsigned short traceid;
char *buf;
unsigned int cnt;
```

The **trsave** kernel subroutine allows device drivers and the AIX kernel to write trace log entries to the **trace** device driver. Application programs should use the **trcunix** kernel subroutine to log trace events. For more information, see "trace_on" in topic 1.2.307, "trcunix" in topic 1.2.308, and "trace" in topic 2.5.29.

Parameters:

traceid High-order 5 bits specify the channel number, and the low-order 11 bits specify the **hookid** for the message. User programs can use only channel number 31.

buf Points to a buffer that contains up to 20 bytes of data for the trace log entry.

cnt Specifies the number of bytes in the buffer pointed to by the **buf** parameter.

If the **trace** device driver has already been opened, and if the channel specified by the **traceid** parameter has been enabled, the log entry is stored in a queue. If there is not enough room in the queue, the entire entry is discarded and a special entry is made to record the fact that it was discarded.

C.6.5 Masking Interrupts

AIX device drivers sometimes need to mask interrupts when in a critical section of code, such as when accessing data that is shared with the **ddintr** routine.

The following are the recommended practices when manipulating interrupt masks:

Do not **spl** downwards except with the **splx** routine. The **splx** routine is used to restore the interrupt mask to the previous level.

Do not mask levels higher than necessary

Match or nest **spl()** and **splx** routines. For example:

```
        .
        .
        .
x = splimp;
        .
        .
        .
splx ( x );
        .
        .
        .
```

To make your code more machine independent, use the descriptiv versions of the mask routines. For example, use the **splhigh** routine instead of the **spl7** routine. Explicit numbers are therefore often used incorrectly. The following kernel subroutines mask and unmask interrupts. They all return the previous **spl** level.

---

splx

```
int splx (level)
int level;
```

The **splx** kernel subroutine masks interrupts at or below the level specified by the **level** parameter, and then returns the current level. Refer to the **/usr/include/sys/i386/intr86.h** file for the appropriate values of the **level** parameter.

Alternatively, you can use the following kernel subroutines:

spl0 Masks interrupts at level 0 and below; all interrupts allowed
spl1 Masks interrupts at level 1 and below
spl2 Masks interrupts at level 2 and below
spl3 Masks interrupts at level 3 and below
spl4 Masks interrupts at level 4 and below

spl5 Masks interrupts at level 5 and below
spl6 Masks interrupts at level 6 and below
spl7 Masks interrupts at level 7 and below; no interrupts allowed at this level. The **spl7** routine should only be used for very short periods of time.

---

splimp

int splimp ()

The **splimp** kernel subroutine is used to protect manipulation of **mbuf** chains for network device drivers in process or interrupt context.

---

splnet

int splnet ()

The **splnet** kernel subroutine is used to mask network software interrupts.

---

splblkio

int **splblkio**()

The **splblkio** kernel subroutine is used to protect buffer manipulation for disk, diskette, and tape device drivers during task-time processing. It disables all interrupts, which would otherwise cause the execution of code that would manipulate data structures associated with block devices, and returns the pre-empted interrupted level. This value is used when restoring interrupts with the **splx()** routine.

---

splhigh

int splhigh()

The **splhigh** kernel subroutine disables all external interrupts and returns the pre-empted interrupted level. This value is used when restoring interrupts with the **splx**( ) routine.

C.6.6 Determining Major and Minor Numbers

The following macros can be called to obtain the major and minor numbers from the **dev** argument passed to device driver entry points.

---

major

```
int major (dev)
dev_t dev;
```

The **major** macro returns the major number portion of the device number.
Parameter:

dev Contains the device's major and minor number.

---

minor

```
int minor (dev)
dev_t dev;
```

The **minor** macro returns the minor number portion of the device number.

Parameter:

dev Contains the device's major and minor number.

C.6.7 Determining Superuser

suser

suser ()

The **suser** kernel subroutine is used to determine whether the effective user ID of the process in context is that of the superuser. This routine can be useful in determining whether special device operations (such as disk formatting) are allowed. If the effective user ID is not that of the superuser, this routine sets the **u.u_error** field to EPERM.

C.7 AIX Kernel Debugger (AIX PS/2)

In order to help with the debugging of new kernel code (such as device drivers) and to help diagnose problems within the kernel or applications, a basic kernel debugger is provided. This debugger can be configured into a kernel, and can be used to examine and modify the state of the machine.

Subtopics
C.7.1 Configuring the Kernel Debugger into a System
C.7.2 Using the Kernel Debugger
C.7.3 Command Descriptions

C.7.1 Configuring the Kernel Debugger into a System

The inclusion of the kernel debugger in a system is controlled by the **kerndbg** parameter in the **sysparms** stanza of the system configuration file **/etc/system**. To include the kernel debugger in a system, add the line:

```
kerndbg = 1
```

to the **sysparms** stanza of your system file. To exclude the kernel debugger from a system, change that line to read:

```
kerndbg = 0
```

Once you have made the necessary changes to the system file, you can build a new kernel with the **newkernel** command. For more information, see the **newkernel** command in *AIX Operating System Commands Reference*.

C.7.2 Using the Kernel Debugger

There are three ways in which the kernel debugger can be invoked:

It can be manually invoked by pressing the key-combinatio **CTRL-ALT-NUM4** on the console keyboard.

It is automatically invoked when a previously set breakpoint i encountered, or after single-stepping. In this case, the debugger prints out the address of the breakpoint that was encountered.

It can be called explicitly within the kernel for debugging in th following way:

```
debugger((intA)0);
```

When the kernel debugger is invoked, it prints out a simple herald of **DEBUG** and prompt you for input with a minus sign. Once you have invoked the kernel debugger, all other system activity ceases. No system activity, other than processing and responding to your commands, takes place until you exit the kernel debugger.

To exit the kernel debugger and resume normal system activities, you can issue the **go** command to the debugger. Note that while you have the system stopped, you may lose data and connections. Incoming serial data may overrun and active network connections may time out.

When you are in the kernel debugger, all command lines should be terminated with a carriage return (the **Enter** key on the console). Input case is ignored. It is not necessary to type the entire name of a command. For instance, the contents of the registers can be displayed by typing either **registers** or just **reg**.

You can obtain a list of available commands by typing **help**. The acceptable abbreviation for each command is shown in upper case. You can obtain usage information on any of the available commands by typing the name of that command, followed by a question mark (?).

All of the commands that accept numeric arguments expect hexadecimal.

Any command that expects an address as an argument accepts any of the following:

A hexadecimal constant

An asterisk followed by a hex constant

A register name

A single quote

That particular address

Indirect that address

Indirect that register

The last address that was actually typed in.

Many of the commands can be modified by flag arguments. Such arguments

are always preceded with a slash /, and can be specified in any order and in any field of the command line.

C.7.3 Command Descriptions

The commands can be divided into four groups:

Basic commands that examine and modify the state of the machin

Commands that aid in the debugging of kernel cod

Commands that display the state of the operating syste

Commands that perform special functions

Subtopics
C.7.3.1 Examining and Modifying Machine State
C.7.3.2 Debugging Kernel Code
C.7.3.3 Displaying Operating System Information
C.7.3.4 Special Functions

C.7.3.1 Examining and Modifying Machine State

The following commands display and modify the contents of memory, the general registers, or the I/O ports:

Command	**Function**
Dump	Dumps the contents of kernel virtual memory. This command displays (in hexadecimal) the contents of a specified area in the kernel's virtual address space. If no address is specified, the dump continues from the address at which the previous dump left off. By default, the dump command displays 128 bytes of memory. The specified area is displayed as bytes, shorts or longs, depending on whether the **/b**, **/s** or **/l** flag is specified. In any case, the contents is also displayed as ASCII characters. An optional second argument specifies the number of lines to dump (each representing 16 bytes).
Enter	Writes data into registers or memory. This command writes data to a registers or into the kernel's virtual address space. If two values are specified, the first is taken to be an address and the second is the value to be stored. If no address is specified, the data is stored at the address following the previous store. If the **/r** flag is specified, the destination address is treated as a register name, and 32 bits of data are stored into the specified register. Otherwise, the amount of data stored is 1, 2 or 4 bytes, according to whether the **/b**, **/s** or **l** flag is specified.
Registers	Dumps out the contents of the registers. This command prints out the contents of the general registers at the time of the interrupt or trap that caused the debugger to be entered. It also prints out the contents of the major debug and control registers if the **/1** flag is specified.
Inport	Reads data from an input port. This command reads a byte or a short from the specified input port. If no address is specified, the same port address used in the previous input command is used. The **/b** and **/s** flags are used to specify whether a byte or a short should be performed.
Outport	Writes data to an output port. This command writes a byte or a short to the specified output port. If two values are specified, it is assumed that the written. If one value is specified it is taken to be the value to be written, and the port number from the previous out command is used. The **/b** and **/s** flags control whether a byte or short should be performed.

C.7.3.2 Debugging Kernel Code

The following commands are used for stack tracing, single stepping, break points, or hexadecimal conversion:

Command	**Function**
BAcktrace	Prints out a kernel stack backtrace. This command prints nested calls, parameters, local variables and saved registers from the time the kernel was entered until the point where the debugger was entered. Normally, this trace starts with the trap or interrupt that caused the debugger to be entered; however, if the **/f** flag (for a full backtrace) is specified, the backtrace starts from the current top of stack (in the **backtrace** routine). Each frame is printed out individually, with the saved previous frame pointer and return pc first, followed by all of the parameters, locals and saved registers that were pushed into the previous frame before the call.
Go	Resumes the interrupted execution. This command causes you to exit from the kernel debugger. The system then resumes its normal activities at the point of interruption.
Trace	Single steps. This command can only be used if the debugger was entered as the result of a breakpoint or a previous single-step operation. It causes one more instruction to be executed and then re-enters the debugger.
BReakpoint	Sets, clears and displays breakpoints. If no arguments are specified, this command prints out a list of the currently set breakpoints. If the **/d** flag is specified, the argument is interpreted as the number of the breakpoint to be disabled. Otherwise, the argument is taken as the address where a new breakpoint should be set. The type of the breakpoint is determined by the following flags: **/r** Sets a hardware read breakpoint **/w** Sets a hardware write breakpoint **/e** Sets a hardware execution breakpoint **/s** Sets a software execution breakpoint **/1** For the **/r**, **/w**, and **/e** flags, set the breakpoint length to one byte **/2** For the **/r**, **/w**, and **/e** flags, set the breakpoint length to two bytes

/4 For the **/r**, **/w**, and **/e** flags, set the breakpoint length to four bytes.

Hardware breakpoints are implemented using the breakpoint hardware on the PS/2. Software execution breakpoints are implemented by replacing the byte at the specified location with a breakpoint instruction.

HEXarith Hexadecimal addition and subtraction.

This command takes two hexadecimal arguments and prints out their sum and difference in both hexadecimal and decimal notation. It handles a surprisingly large portion of your hexadecimal arithmetic needs including hexadecimal to decimal conversion.

C.7.3.3 Displaying Operating System Information

The following commands display important information regarding the state of the operating system:

Command	Function
Gdt	Prints the contents of the global descriptor table.
Ldt	Prints the contents of the local descriptor table. The **Gdt** and **Ldt** commands print the entire contents of the Global or Local descriptor tables--descriptor numbers, type, length, address and flags.
MEMfree	Displays information about the free memory pool.
Process	Prints information from the process table. If the **/a** flag is specified, only information on active processes is printed. This command can also take one or two numeric arguments. The first argument specifies which process table slot is to be printed. The second argument specifies the number of process table slots to be printed. If the second argument is not specified, all processes are printed.
PVseg	Prints out **procvseg** structures. With no arguments, **pvseg** prints the **procvseg** structure of the process in context. If an argument is given, it is used as an address of a specific **procvseg** structure to be displayed.
SWAPfree	Displays information about the available swap space.
Version	Prints system version information. This command prints a string indicating when and where the base system was built.
VSeg	Prints **vseg** structures. With no arguments, the **vseg** command prints out all of the **vseg** structures on the **vseg** busylist. If an argument is given, it is used as an address of a specific **vseg** to be displayed.

C.7.3.4 Special Functions

The following commands perform special functions:

Command	**Function**
Help	Displays a help menu for debugger commands.
REBoot	Reboots the system. This command prompts you about rebooting the system. If you respond with a **y**, the system halts immediately without flushing system buffers to disk (which would synchronize disk and buffer versions).

C.8 Driver Configuration and Initialization

This section describes the following aspects of driver configuration and initialization on AIX:

Adding a device driver into the AIX kerne

Driver configuration component

Adding devices support for device driver

Passing parameters to a customize helpe

Passing parameters to a special processing routin

Adding possible choices for the devices command to displa

Adding descriptions for the devices command to display

Subtopics

C.8.1 Adding a Device Driver into AIX Kernel

The **newkernel** command rebuilds the AIX kernel. This command links a device driver into the AIX kernel when one of the following is true:

The **mandatory** flag in the driver's **/etc/master** stanza is set to true

There is a stanza associated with the device driver in the **/etc/system** file.

If the **mandatory** flag is set to true, the driver is always included into the kernel by the **newkernel** command. Only certain device drivers such as the console and fixed disk device drivers that are critical to the running of AIX should have **mandatory** set to true.

If you do not want to configure the device driver, you can add the driver into the AIX kernel by performing the following steps:

1. Compile the device driver using the following command:

```
cc -c -DKERNEL -Di386 -I/usr/include/sys driver.c
```

   where **driver.c** is a C source file for your device driver.

2. Archive the resulting **.o** files into the **386lib** kernel archive by entering this command:

```
ar -rv /usr/sys/386/386lib.a driver.o
```

3. Place a stanza for the device driver into the **/etc/master** file, and make sure you have picked a unique major number. If your device driver is autoconfigured and supports both character and block devices, the stanza that you add to the **/etc/master** file should look something like the following:

```
driver: type = dev
        routines = init,open,close,read,write,ioctl,intr,select,strategy
        softcfg = TRUE
        major = 45
        maxminor = 1
        prefix = dd
        struct = ddtab
```

4. Add a stanza similar to the following to the **/etc/system** file:

```
spdrvr0:
        driver = driver
        nospecial = true
        noshow = true
```

5. Rebuild the AIX kernel by entering the following commands:

```
cd /usr/sys
newkernel
```

6. Reboot the machine by entering:

```
reboot
```

7. When the system reboots, add the special file associated with the device by entering:

```
mknod /dev/rspdrv0 c 45 0
mknod /dev/spdrv0 b 45 0
```

8. Write an AIX application that issues system calls to your device.

C.8.2 Driver Configuration Components

The following is a description of the important elements in the configuration of PS/2 device drivers:

Configuration Files
The following is a brief summary of PS/2 configuration files.

/etc/master Describes available device drivers, specified system parameters, and configured system devices. Device drivers are described by their name, entry points, major device number, driver type, and associated table declarations. System parameters are described by the parameter value, the default value, symbol type (string or numeric) and identifies the kernel variables to be defined or patched while the **/etc/config** file is executed. Device class requirements and default values detail system devices.

/etc/system Enumerates what AIX device drivers are to be configured into the AIX kernel. Each device that is added to the **/etc/system** file via the **installp**, **updatep**, or **devices** command contains a stanza. There can be more than one stanza per device driver in the **/etc/system** file.

The **/etc/system** file also overrides default system parameter values and designates system devices.

/etc/ddi/*.ddi
Contains device-dependent information. Each device that you configure has this device-dependent information attribute file associated with it.

/etc/ddi/*.kaf
Contains keyword attribute files which have the input checking mechanisms if a device has displayable configuration information.

/etc/predefined
Contains stanza skeletons that are added to the **/etc/system** file by the **devices** command.

API Contains a series of subroutines (see the **cfgadev**, **cfgaply**, **cfgcadsz**, **cfgcdlsz**, **cfgcopsf**, **cfgcrdsz**, and **cfgddev** routines in Volume I) that allow programs to manipulate attribute files, add and delete devices, and apply information into the system kernel. The following is the list of configuration programs that use the API:

/etc/devices

/etc/installp

/etc/updatep

/etc/penable, /etc/pstart, /etc/pdelay, /etc/pdisable,

/etc/phold, and **/etc/pshare**.

When installing a new driver into AIX via the **installp** command, the API searches the **/etc/master** file and assign the driver a unique major number.

When a stanza is added to the **/etc/system** file by the API, the following occurs:

1. The API read the device's stanza out of the **/etc/predefined** file.

2. The **devices** command assigns the stanza a unique minor number, name, slot and port.

3. The API adds the stanza to the **/etc/system** file.

4. If the driver is in the kernel, the **osconfig** command is used to add the special file associated with the device. For more information on the **osconfig** command, refer to the *AIX Commands Reference*. This command invokes the driver's customization helper to issue **ioctl** system calls to the device driver to configure it.

5. If the driver is not in the kernel, the **osconfig** command is used to add the special files and not to call the customize helper. The **devices** command rebuilds the kernel upon exit.

6. The driver's special processing routine is called if the **osconfig** command successfully adds the device.

When a device is deleted, the following occurs:

1. The device's stanza is removed from the **/etc/system** file.

2. The device's API invokes **osconfig** command to inform the device driver that a minor number has been deleted from device. This command calls the customization helper to notify the device driver that the device is being deleted, usually via **ioctl** system calls, and then deletes the special file associated with the device.

3. The API invokes the special processing routine of the driver indicating that the device has been deleted.

4. If all stanzas associated with a device driver are deleted from the **/etc/system** file and the **mandatory** flag is not set to true in the **/etc/master** file, the AIX kernel is rebuilt.

When a device is changed, one or more of the following occurs:

The special processing routine of the device is invoked

The kernel is rebuilt

Devices such as the fixed disk, diskette, and system console are not user configurable by API. Special devices associated with the diskette drives, including those for the external

5.25-inch external diskette drive, always exist in the AIX base, whether or not they are used. Fixed disks are configured by the **minidisks** command. Stanzas exist in the **/etc/system** file for each minidisk, fixed disk, diskette drive, and the system console.

The API is stored in the **/lib/librts.a** file.

osconfig Performs the following functions:

Creates or deletes special files when the user adds or deletes a device.

If there is a customization helper associated with the device, as specified by the **config** parameter in the device's **/etc/master** stanza, the customization helper is called during adding, deleting, starting up, or shutting down.

The **osconfig** command is invoked every time AIX is booted from single to multi user mode, in the **/etc/init.dir/Singl2multi** file. For more information, see the **osconfig** command in the *AIX Operating System Commands Reference*.

Customization helper
Interprets the information in the device configuration files and then configures the AIX device drivers by issuing **ioctl** routines.

Installp, updatep
Add code to the AIX kernel, update the system configuration files, and rebuild the AIX kernel. These commands are used by developers of licensed program products (LPPs).

pstart, pdelay, pdisable, penable, phold, pshare
Allows a user to manipulate the status of a terminal device or pseudo-terminal.

device Allows the user to add and delete special devices and reconfigure device-specific parameters

/usr/sys/newkernel
Builds and installs the AIX kernel. While rebuilding the kernel, the **newkernel** command invokes a program called **/etc/config** that generates a configuration summary, a file containing device switch tables, external variable definitions, and other pertinent kernel information. The configuration summary defines what device drivers are added to the kernel and the values of system parameters.

The **newkernel** command is invoked when an API-based application calls the **cfgaply** subroutine.

/etc/config Processes the **/etc/master** and **/etc/system** files to build a machine-independent **conf.c** file and a configuration summary, thereby pulling drivers into or out of the kernel linkage edit.

The configuration summary is used to patch the **/usr/sys/conf.o** file so that the AIX kernel can be rebuilt without a C

compiler.

conf.c Contains the following kernel configuration information:

devsw The device switch table. A table of configured device drivers, indexed by the major device number. Once a device's **ddinit** routine is placed into the device switch table, it is configured into the AIX kernel. If the device driver is autoconfigured, then after the rest of the driver's entry points are placed into the switch table, subsequent system calls or socket calls enter the device driver's entry points or procedure handles.

spinitsw Configures non-driver packages; contains initialization entry points only.

gensw The system configuration summary. A table of all configured devices and drivers in the AIX kernel.

System device designations
Specifies the initialization of major and minor numbers for each system device. Examples of system devices are the root file system, the local file system, the swap device, and so forth.

Parameter values
Definitions for each parameter with the default or overridden value.

Variable definitions
Important system variable definitions. Most definitions come from either the **/usr/include/sys/space.h** or the **/usr/include/sys/i386/sufcfg386.h** file.

Static table allocations
Important static table allocations. Most tables are initialized in the **/usr/include/sys/space.h** or **/usr/include/sys/i386/sufcfg386.h** file.

Special Processing

An optional program that the **devices** command invokes when adding or deleting a device. There are special processing routines built into the API when a printer or **tty** port is added.

Special processing routines can be used to update the **/etc/environment** file, to add a variable concerning the device driver for every AIX user, and to add an additional process that is run during system boot by editing the **/etc/init.dir/Singl2multi** file.

Special processing routines are only called during the adding, deleting or changing of the device, not during system boot.

AIX Device Drivers

Device drivers that are configured via the **devices** command contain support for **ddioctl** routines that customization

helpers issue to configure the device driver.

A device driver can be both autoconfigured and configured by the **devices** command. An autoconfigured device driver determines the hardware information from the POS data base while customization helpers use **ioctl** system calls to configure the software parameters of the device driver.

Network device drivers do not require a customization helper.

See Figure C-12 for an illustration of configuration components.

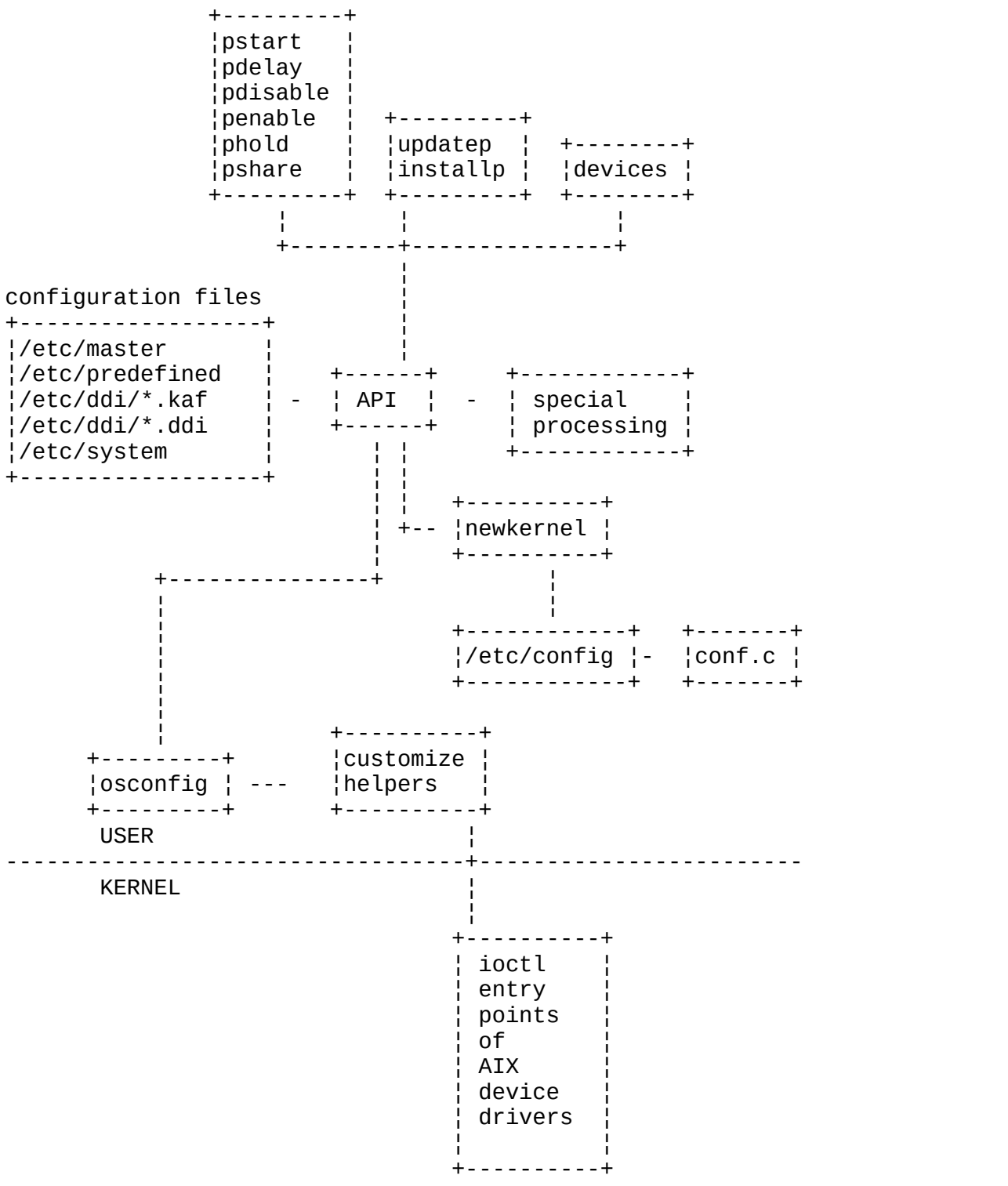


Figure C-12. AIX Configuration Overview

C.8.3 Adding Devices Support for Device Drivers

It is possible to add devices support for device drivers. Suppose you have a character device driver, **sampldrv**, that you wish to configure with the **devices** command. In addition, suppose that the attributes associated with this particular driver are as follows:

One configured parameter is **cfgprm**, that can be any value from 1 to 5.

The customization helper associated with the driver is calle "samphlpr".

The device is not multiplexed

The device has **ddopen**, **ddclose**, **ddread**, **ddwrite**, **ddioctl**, and **ddselect** entry points and an interrupt handler.

The device is autoconfigured, supporting up to four adapters with a adapter ID of 0xE2D2. Each adapter is represented by a minor number.

There is no special processing routine associated with the devic driver.

To allow this device driver to be reconfigured via the **devices** command, perform the following steps:

1. Create a stanza in the **/etc/master** file for the device driver:

```
sampdrv:        type = dev
        config = samphlpr
        routines = init,open,close,read,write,ioctl,intr,select
        softcfg = TRUE
        nocount = TRUE
        char = TRUE
        mpx = FALSE
        prefix = sampdrv
        major = 48
        maxminor = 4
```

2. Update the **/etc/predefined** file as follows:

   a. Add the following lines to the **ports** stanza:

```
        samp1 = 0
* sample port 1
        samp2 = 1
* sample port 1
        samp3 = 2
* sample port 1
        samp4 = 3
* sample port 1
```

   Each of the four adapters can be thought of a port. Therefore, the first adapter's port name is **samp1** and its minor number is 0.

   b. Each adapter ID has a range of ports. For our device, we would

add the following lines to the **adapts** stanza:

```
e2d2 = samp1-4
* IBM PS/2 Sample Adapter
```

3. Add the template for the stanza to be added into the **/etc/system** file:

```
sampdrv:
* IBM PS/2 Sample Adapter
        name = samp
        driver = sampdrv
        minor = c
        kaf_file = /etc/ddi/samp.ddi
        kaf_use = ksamp
        file = /etc/ddi/samp.kaf
        use = dsamp
        noddi = false
        noduplicate = false
        dtype = adapters
* IBM PS/2 Adapters
        noshow = false
        dname = sampc
        pflag = true
* sample port 1
        port = samp1-4
        slot = 0
        noipl = false
```

4. Create a **ddi** file in **/etc/ddi**:

```
default:
* defaults for Sample Device driver
sysadd = o
sysdel = o
cfgprm = 2
* The one changeable parameter   * 1 - 5

sampcsamp1:

sampcsamp2:

sampcsamp3:

sampcsamp4:
```

5. Create a **kaf** file in **/etc/ddi**:

```
default:
        smf_file = /etc/ddi/samp.kaf

cfgprm:
        syschg = none
        vtype = 3
```

```
display = true
type = I
range = 1,5,1
```

6. Write a customization helper as follows:

```
#include <stdio.h>
#include <errno.h>
#include <cfgcdefs.h>
#include <fcntl.h>
#include <sys/sampdrv.h>         /* defines for IOCTL's to d.d. */

#ifdef NLS
#define STRTOK NLstrtok
#else
#define STRTOK strtok
#endif

char *STRTOK();

main( argc, argv )
        int argc;
        char *argv[]
{
        int i;
        int add = 0;
        int delete = 0;
        char *special;                  /* pointer to the special file
                                                name */
        char temp_parm[100];
        char temp_value[100];
        int val_cfgprm;                 /* the value of cfgprm */
        int fd;
        extern int errno;
        extern int sys_nerr;
        char chgprm[100];    /* pointer to the one lone parameter */
        int ret_code = 0;

        /* determine if we are adding or deleting */
        if ( !strcmp ( argv[1], "-a" ) ||
                !strcmp ( argv[1], "-startup" ) )
                add = 1;

        if ( !strcmp ( argv[1], "-d" ) )
                delete = 1;

        /* get the special file name */
        special = STRTOK ( argv[2], ":" );

        /* get the value for cfgprm */
        for (i=4; i < argc; i++ )
        {
                temp_parm = STRTOK ( argv[i], "=" );
                temp_value = STRTOK ( NULL, "" );
                if ( strstr(temp_parm,"cfgprm") != NULL )
                {
                        strcpy(cfgprm,temp_value);
                        break;
                }
        }
```

```
        /* get the value for cfgprm */
        val_cfgprm = atoi ( cfgprm );

        /* if this is an add then issue the ioctl to tell
                the device driver the value of cfgprm */
        if ( add )
        {
                /* open the special file */
                if ( (fd = open (special, O_RDWR )) != -1 )
                {
                    if ( ioctl ( fd, SAMP_ADD, val_cfgprm, 0 ) == -1 )
                    {
                        fprintf( stderr,
                                "Error %d performing sample initializati
    errno );
                        return (errno);
                    }
                    close (fd);
                }

                else
                {
                        fprintf( stderr," Could not open
                                 %s errno = %d-n", special_file,
                                errno );
                        ret_code = errno;
                }
        }

        /* if this is a delete then just return */
        if ( delete )
        {
                /* do nothing for this device driver */
                ret_code = 0;
        }
        return (ret_code);
}
```

C.8.4 Parameters Passed to a Customization Helper

The following parameters can be passed to a customization helper:

argv[1] The customization flag which has one of the following values:

- **-a** Add the device. Called after the device is added with the **devices** command.
- **-d** Delete the device. Called after the device is deleted with the **devices** command.
- **-startup** Startup the device. Called during system boot. The customization helper should perform similar operations to that of the **a** flag.

argv[2] The special file name, such as **tty0** or **3270c0**. The special file name is also the name of the stanza in the **/etc/system** file.

argv[3] The driver name, as specified by the driver parameter in the device's **/etc/system** stanza.

argv[4] - argv[argc-1]
The rest of the arguments passed to the customization helper are:

All of the **keyword = value** pairs from the **/etc/system** and **/etc/master** files and **ddi** stanzas for the device.

The **keyword = value** pairs from the **default** stanzas in the **ddi** file for the device.

An example illustrating how the arguments are passed follows:

```
argv[4] = major=45
argv[5] = maxminor=1
argv[6] = prefix=drvr

        .
        .
        .
```

The customization helper parses **argv[4]** to **argv[argc-1]**, fills out the appropriate device driver data structures, and then issues the **ioctl** routines to configure the device driver.

C.8.5 Parameters Passed to a Special Processing Routine

The following parameters are passed to the special processing routine from the API:

argv[1] The full path of the master file, such as **/etc/master**

argv[2] The full path of the system file, such as **/etc/system**

argv[3] The special file name, such as **tty0** or **3270c0**. The device name is also the name of the stanza in the **/etc/system** file.

argv[4] Customization flag, having one of the following values:

- **-a** Add the device. Called after the device is added with the **devices** command
- **-as** Add the special file associated with the device but do not call the customization helper. Handle this flag just as you would handle **-a**.
- **-d** Delete the device. Called after the device is deleted with the **devices** command.

C.8.6 Adding Descriptions for Device Command to Display

Most screens displayed by the **devices** command contain a column for descriptions. You must provide the descriptions in the configuration files in order for them to be displayed.

Descriptions fall into two categories:

Descriptions for keywords in the **/etc/system**, **/etc/master**, and **/etc/predefined** files. To add a descriptive phrase for a keyword in one of these files, put the description in a comment line following the keyword it describes.

The following example shows how a comment for the 5152 printer might appear in the **/etc/predefined** file.

```
5152:
* IBM PC Graphics Printer (5152)
        . . .
        noddi = false
        dtype = printer
* Printer
```

Descriptions for keywords in the **ddi** files. There are two methods for providing descriptions in the **/etc/ddi** files. They both produce the same results, but the second method can save some file space.

- The first method is the same as descriptions for the **/etc/system**, **/etc/master**, and **/etc/predefined** files. Put the description in a comment line following the keyword it describes.

  The following is an example of descriptive text in the **/etc/predefined** file:

```
default:
        . . .
        lpi = 6
* Lines per Inch              * 6, 8
        ep = no
* Emphasized Print            * yes, no
```

- The second method is to combine all the descriptions into the **/etc/ddi/descriptions** file. If a description line following the keyword (as described in the first method) is not found, the **devices** command uses the keyword as the key to search the **/etc/ddi/descriptions** file for a description to display.

C.8.7 Adding Choices for the Devices Command to Display

Some screens provided by the **devices** command contain a "Possible Choices" column heading. Under that heading are the valid choices for the associated **ddi** keyword. You must provide the possible choices in the configuration files in order for them to be displayed. There are two methods for providing the possible choices:

1. Incorporate the choices with the keyword descriptions as explained under "Descriptions" in the **/etc/ddi** file. In the same line as the keyword description, insert an asterisk (*) followed by the possible choices. As seen in the above example, the second set of comments (**6, 8** and **yes, no**) are the valid choices for the "Possible Choices" column.

2. Combine all valid options into one file called **/etc/ddi/options**. This file must follow the specific format described under "options" in topic 2.3.43. If the **devices** command does not find a description line following the keyword as explained in method one under Descriptions in **/etc/ddi**, the **devices** command looks for the **opts** keyword in the **kaf** file stanza for the keyword. The **devices** command then uses the **opts** keyword to generate a key to search on in the **/etc/ddi/options** file. The **devices** command then displays the choices column. The format of the **/etc/ddi/options** file is important.

D.0 Appendix D. Glossary

access. To obtain data from or put data in storage.

access permission. A group of designations that determine who can access a particular AIX file and how the user may access the file.

account. The log in directory and other information that give a user access to the system.

activity manager. A collection of system-supplied tasks allowing users to manage their activities. Provides the ability to list current activities (Activity List) and to begin, cancel, hide, and activate activities.

All Points Addressable (APA) display. A display that allows each pel to be individually addressed. An APA display allows for images to be displayed that are not made up of images predefined in character boxes. Contrast with **character display**.

allocate. To assign a resource, such as a disk file or a diskette file, to perform a specific task.

alphabetic. Pertaining to a set of letters a through z.

alphanumeric character. Consisting of letters, numbers and often other symbols, such as punctuation marks and mathematical symbols.

American National Standard Code for Information Interchange (ASCII). The code developed by ANSI for information interchange among data processing systems, data communications systems, and associated equipment. The ASCII character set consists of 7-bit control characters and symbolic characters.

American National Standards Institute. An organization sponsored by the Computer and Business Equipment Manufacturers Association for establishing voluntary industry standards.

application. A program or group of programs that apply to a particular business area, such as the Inventory Control or the Accounts Receivable application.

application program. A program used to perform an application or part of an application.

argument. Numbers, letters, or words that change the way a command works.

ASCII. See **American National Standard Code for Information Interchange**.

asynchronous transmission. In data communication, a method of transmission in which the bits included in a character or block of characters occur during a specific time interval. However, the start of each character or block of characters can occur at anytime during this interval. Contrast with **synchronous transmission**.

attribute. A characteristic. For example, the attribute for a displayed field could be blinking.

authorize. To grant to a user the right to communicate with, or make use of, a computer system or display station.

auto carriage return. The system function that places carriage returns automatically within the text and on the display. This is accomplished by moving whole words that exceed the line end zone to the next line.

backend. The program that sends output to a particular device. There are two types of backends: friendly and unfriendly.

background process. (1) A process that does not require operator intervention that can be run by the computer while the work station is used to do other work. (2) A mode of program execution in which the shell does not wait for program completion before prompting the user for another command.

backup copy. A copy, usually of a file or group of files, that is kept in case the original file or files are unintentionally changed or destroyed.

backup diskette. A diskette containing information copied from a fixed disk or from another diskette. It is used to restore information in case the original information becomes unusable.

bad block. A portion of a disk that can never be used reliably.

base address. The beginning address for resolving symbolic references to locations in storage.

base name. The last element to the right of a full path name. A filename specified without its parent directories.

batch printing. Queueing one or more documents to print as a separate job. The operator can type or revise additional documents at the same time. This is a background process.

batch processing. A processing method in which a program or programs execute with little or no operator action. This is a background process.

Contrast with **interactive processing**.

binary. (1) Pertaining to a system of numbers to the base two; the binary digits are 0 and 1. (2) Involving a choice of two conditions, such as on-off or yes-no.

bit. From BInary digiT. Either of the binary digits 0 or 1 used in computers to store information. See also **byte**.

block. (1) A group of records that is recorded or processed as a unit. Same as **physical record**. (2) In data communication, a group of records that is recorded, processed, or sent as a unit. (3) A physical block in AIX is 4096 bytes long. (4) A logical block in AIX is 1024 bytes long.

block file. A file listing the usage of blocks on a disk.

block special file. A special file that provides access to an input or output device that is capable of supporting a file system. See also **character special file**.

bootstrap. A small program that loads larger programs during system initialization.

branch. In a computer program an instruction that selects one of two or more alternative sets of instructions. A conditional branch occurs only when a specified condition is met.

breakpoint. A place in a computer program, usually specified by an instruction, where execution may be interrupted by external intervention or by a monitor program.

buffer. (1) A temporary storage unit, especially one that accepts information at one rate and delivers it at another rate. (2) An area of storage, temporarily reserved for performing input or output, into which data is read, or from which data is written.

burst pages. On continuous-form paper, pages of output that can be separated at the perforations.

byte. The amount of storage required to represent one character; a byte is 8 bits.

call. (1) To activate a program or procedure at its entry point. Compare with **load**.

callouts. An AIX kernel parameter establishing the maximum number of scheduled activities that can be pending simultaneously.

canonical processing. Processing that occurs according to a defined set of rules.

cancel. To end a task before it is completed.

carriage return. (1) In text data, the action causing line ending formatting to be performed at the current cursor location followed by a line advance of the cursor. Equivalent to the carriage return of a typewriter. (2) A keystroke generally indicating the end of a command line.

case sensitive. Able to distinguish between uppercase and lowercase letters.

character. A letter, digit, or other symbol.

character display. A display that uses a character generator to display predefined character boxes of images (characters) on the screen. This kind of display cannot address the screen any less than one character box at a time. Contrast with **All Points Addressable display**.

character key. A keyboard key that allows the user to enter the character shown on the key. Compare with **function keys**.

character position. On a display, each location that a character or symbol can occupy.

character set. A group of characters used for a specific reason; for example, the set of characters a printer can print or a keyboard can support.

character special file. A special file that provides access to an input or output device. The character interface is used for devices that do not use block I/O. See also **block special file**.

character string. A sequence of consecutive characters.

character variable. The name of a character data item whose value may be assigned or changed while the program is running.

child. (1) Pertaining to a secured resource, either a file or library, that uses the user list of a parent resource. A child resource can have only one parent resource. (2) In the AIX Operating System, child is a **process** spawned by a parent process that shares resources of parent process. Contrast with **parent**.

C language. A general purpose programming language that is the primary language of the AIX Operating System.

class. Pertaining to the I/O characteristics of a device. AIX devices are classified as block or character.

client. A computer or process that accesses the data, services, or resources of another computer or process in a network.

close. (1) To end an activity and remove that window from the display. (2) To finalize I/O operations on a file.

code. (1) Instructions for the computer. (2) To write instructions for the computer; to **program**. (3) A representation of a condition, such as an error code.

code page. (1) An assignment of graphic characters and control function meanings to all code points. (2) Arrays of code points representing characters that establish ordinal sequence (numeric order) of characters. AIX uses 256-character code pages.

code point. (1) A 1-byte code representing one of 256 potential characters. (2) A 1- or 2-byte representation of a character. A byte can contain a single-shifted bit that indicates that the second byte is a part of the same code point, and indicates the code page of the character. The second byte (only byte in the case of a 1-byte character) places the character in the code page array.

code segment. See **segment**.

collating sequence. The sequence in which characters are ordered within the computer for sorting, combining, or comparing.

color display. A display device capable of displaying more than two colors and the shades produced via the two colors, as opposed to a monochrome display.

color expansion operation. A graphics programming operation that occurs automatically when the source pixel map data area contains only one bit per pixel, and the destination pixel map data area is a color display adapter buffer frame defined to have more than one bit per pixel.

column. A vertical arrangement of text or numbers.

column headings. Text appearing near the top of columns of data for the purpose of identifying or titling.

command. A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

command interpreter. A program (such as the Bourne or C shell) that reads text lines representing commands and invokes the corresponding command.

command line. (1) The area of the screen where commands are displayed as they are typed. (2) The contents of such a line.

command line editing keys. Keys for editing the command line.

command programming language. Facility that allows programming by the combination of commands rather than by writing statements in a conventional programming language.

command word. The name of the 16-bit units used for storing graphic primitive strings (GPS). The first command word determines the primitive type and sets the length of the string. Subsequent command words contain information in multiples of quid, or, four bits of data.

compile. (1) To translate a program written in a high-level programming language into a machine language program. (2) The computer actions required to transform a source file into an executable object file.

compress. (1) To move files and libraries together on disk to create one continuous area of unused space. (2) In data communications, to delete a series of duplicate characters in a character string.

concatenate. (1) To link together. (2) To join two character strings.

condition. An expression in a program or procedure that can be evaluated to a value of either true or false when the program or procedure is running.

configuration. The group of machines, devices, and programs that make up a computer system. See also **system customization**.

configuration file. A file that specifies the characteristics of a system or subsystem, for example, the AIX queueing system.

consistent. Pertaining to a file system, without internal discrepancies.

console. (1) The main AIX display station. (2) A device name associated with the main AIX display station.

constant. A data item with a value that does not change. Contrast with **variable**.

context search. A search through a file whose target is a character string.

control block. A storage area used by a program to hold control information.

control character. (1) A character, occurring in a particular context, that initiates, modifies, or stops any operation that affects the recording, processing, transmission, or interpretation of data (such as carriage return, font change, and end of transmission). (2) A non-printing character that performs formatting functions in a text file.

control code. A code point and its assigned control function meaning; for example, "end of transmission". Control codes do not have graphical representations. For 7-bit codes such as ASCII, the first 32 code points are reserved for control purposes. See also *single-shift control*.

control program. Part of the AIX Operating System that determines the order in which basic functions should be performed.

controlled cancel. The system action that ends the job step being run, and saves any new data already created. The job that is running can continue with the next job step.

copy. The action by which the user makes a whole or partial duplicate of already existing data.

coupler. A device connecting a modem to a telephone network.

crash. An unexpected interruption of computer service, usually due to a serious hardware or software malfunction.

current directory. The directory that is active, and can be displayed with the **pwd** command.

current line. The display line on which the cursor is located.

current working directory. See **current directory**.

cursor. (1) A movable symbol (such as an underline) on a display, used to indicate to the operator where the next typed character will be placed or where the next action will be directed. (2) A marker that indicates the current data access location within a file.

cursor movement keys. The directional keys used to move the cursor.

customize. To describe (to the system) the devices, programs, users, and user defaults for a particular data processing system.

cylinder. All fixed disk or diskette tracks that can be read or written without moving the disk drive or diskette drive read/write mechanism.

daemon. See **daemon process**.

daemon process. A process begun by the root or the root shell that can be stopped only by the root. Daemon processes generally provide services that must be available at all times such as sending data to a printer.

data block. See **block**.

data communications. The transmission of data between computers, or remote devices or both (usually over long distance).

data link. The equipment and rules (protocols) used for sending and receiving data.

data stream. All information (data and control information) transmitted over a data link.

debug. (1) To detect, locate, and correct mistakes in a program. (2) To find the cause of problems detected in software.

default. A value that is used when no alternative is specified by the operator.

default directory. The directory name supplied by the operating system if none is specified.

default drive. The drive name supplied by the operating system if none is specified.

default value. A value stored in the system that is used when no other value is specified.

delete. To remove. For example, to delete a file.

dependent work station. A work station having little or no stand-alone

capability, that must be connected to a host or server in order to provide any meaningful capability to the user.

device. An electrical or electronic machine that is designed for a specific purpose and that attaches to your computer, for example, a printer, plotter, disk drive, and so forth.

device driver. A program that operates a specific device, such as a printer, disk drive, or display.

device manager. Collection of routines that act as an intermediary between device drivers and virtual machines for complex interfaces. For example, supervisor calls from a virtual machine are examined by a device manager and are routed to the appropriate subordinate device drivers.

device name. A name reserved by the system that refers to a specific device.

diagnostic. Pertaining to the detection and isolation of an error.

diagnostic aid. A tool (procedure, program, reference manual) used to detect and isolate a device or program malfunction or error.

diagnostic routine. A computer program that recognizes, locates, and explains either a fault in equipment or a mistake in a computer program.

digit. Any of the numerals from 0 through 9.

directory. A type of file containing the names and controlling information for other files or other directories.

disable. To make nonfunctional.

discipline. Pertaining to the order in which requests are serviced, for example, first-come-first-served (fcfs) or shortest job next (sjn).

discriminated unions. An XDR discriminated union is a set of data composed of a discriminant and another data type. The discriminant is an enumeration. The other data type is selected from a set of prearranged types according to the value of the discriminant. The component types are called arms of the union. The discriminated union is encoded starting with the discriminant followed by the arm.

disk I/O. Fixed-disk input and output.

diskette. A thin, flexible magnetic plate that is permanently sealed in a

protective cover. It can be used to store information copies from a fixed disk or another diskette.

diskette drive. The device used to read and write information on diskettes.

display device. An output unit that gives a visual representation of data.

display screen. The part of the display device that displays information visually.

display station. A device that includes a keyboard from which an operator can send information to the system and a display screen on which an operator can see the information sent to or received from the computer.

distributed file system. A file system whose files, directories, and other components are stored on different sites in a particular cluster.

distributed operating system. An operating system where multiple machines cooperate to seem like one machine.

distributed processing. Results when a user involves multiple cluster sites in a single operation--for example, by editing a remote file and starting a task on another cluster site using the **on, fast, fastsite,** and **migrate** commands.

Distributed Services (DS). A licensed program that allows you to share files with other AIX systems in a network. You can mount the file systems located on other AIX systems to create file trees that are independent of the file systems.

dump. (1) To copy the contents of all or part of storage, usually to an output device. (2) Data that has been dumped.

dump diskette. A diskette that contains a dump or is prepared to receive a dump.

dump formatter. Program for analyzing a dump.

EBCDIC. See **extended binary-coded decimal interchange code**.

EBCDIC character. Any one of the symbols included in the 8-bit EBCDIC set.

edit. To modify the form or format of data.

edit buffer. A temporary storage area used by an editor.

editor. A program used to enter and modify programs, text, and other types of documents and data.

effective root directory. The point where a system starts when searching for a file. Its path name begins with /(slash).

emulation. Imitation; for example, when one computer imitates the characteristics of another computer.

enable. To make functional.

enter. To send information to the computer by pressing the **Enter** key.

entry. A single input operation on a work station.

environment. The settings for shell variables and paths associated with each process. These variables can be modified later by the user.

error-correct backspace. An editing key that performs editing based on a cursor position; the cursor is moved one position toward the beginning of the line, the character at the new cursor location is deleted, and all characters following the cursor are moved one position toward the beginning of the line (to fill the vacancy left by the deleted element).

escape character. A character that suppresses the special meaning of one or more characters that follow.

Ethernet. A physical medium through which computers in the same or different clusters can communicate and share files.

exit value. A numeric value that a command returns to indicate whether it completed successfully. Some commands return exit values that give other information, such as whether a file exists. Shell programs can test exit values to control branching and looping.

expression. A representation of a value. For example, variables and constants appearing alone or in combination with operators.

extended binary-coded decimal interchange code (EBCDIC). A set of 256 eight-bit characters.

extended character. A graphic character other than a 7-bit ASCII

character. An extended character can be a 1-byte code point with the eighth bit set or a 2-byte code point.

feature. A programming or hardware option, usually available at an extra cost.

field. (1) An area in a record or panel used to contain a particular category of data. (2) The smallest component of a record that can be referred to by a name.

FIFO. See **first-in-first-out**.

file. A collection of related data that is stored and retrieved by an assigned name.

file name. The name used by a program to identify a file. See also **label**.

filename. In DOS, that portion of the file name that precedes the extension.

file specification (filespec). The name and location of a file. A file specification consists of a drive specifier, a path name, and a file name.

file system. The collection of files and file management structures on a physical or logical mass storage device, such as a diskette or minidisk.

filetab. An AIX kernel parameter establishing the maximum number of files that can be open simultaneously.

filter. A command that reads standard input data, modifies the data, and sends it to standard output.

first-in-first-out (FIFO). A named permanent pipe. A FIFO allows two unrelated processes to exchange information using a pipe connection.

fixed disk. A flat, circular, nonremoveable plate with a magnetizable surface layer on which data can be stored by magnetic recording.

fixed-disk drive. The mechanism used to read and write information on fixed disk.

flag. A modifier that appears on a command line with the command name that defines the action of the command. Flags in the AIX Operating System almost are always preceded by a dash.

font. A family or assortment of characters of a given size and style.

flattened character. An ASCII character created by translating an extended character to the ASCII character most like it. The code point information is lost and the character cannot be retranslated to an extended character. For example, a **c cedilla** would be flattened to a plain **c**.

foreground. A mode of program execution in which the shell waits for the program specified on the command line to complete before returning your prompt.

format. (1) A defined arrangement of such things as characters, fields, and lines, usually used for displays, printouts, or files. (2) The pattern which determines how data is recorded.

formatted diskette. A diskette on which control information for a particular computer system has been written but which may or may not contain any data.

free list. A list of available space on each file system. This is sometimes called the free-block list.

free-block list. See **free list**.

full path name. The name of any directory or file expressed as a string of directories and files beginning with the root directory.

function. A synonym for procedure. The C language treats a function as a data type that contains executable code and returns a single value to the calling function.

function keys. Keys that request actions but do not display or print characters. Included are the keys that normally produce a printed character, but when used with the code key produce a function instead. Compare with **character key**.

generation. For some remote systems, the translation of configuration information into machine language.

Gid. See **group number**.

global. Pertains to information available to more than one program or subroutine.

global action. An action having general applicability, independent of the

context established by any task.

global character. The special characters * and ? that can be used in a file specification to match one or more characters. For example, placing a ? in a file specification means any character can be in that position.

global search. The process of having the system look through a document for specific characters, words, or groups of characters.

global variable. A symbol defined in one program module, but used in other independently assembled program modules.

GPS. See *graphic primitive string*.

graphic character. A character that can be displayed or printed.

group name. A name that uniquely identifies a group of users to the system.

graphic primitive string (GPS). The format used for storing graphics file data. A GPS is composed of up to five types of graphical data: comments, lines, arcs, text, and hardware.

group number (Gid). A unique number assigned to a group of related users. The group number can often be substituted in commands that take a group name as an argument.

hardware. The equipment, as opposed to the programming, of a computer system.

header. Constant text that is formatted to be in the top margin of one or more pages.

header label. A special set of records on a diskette describing the contents of the diskette.

here document. Data contained within a shell script (also called **inline input**).

hexadecimal. Pertaining to a system of numbers to the base sixteen; hexadecimal digits range from 0 (zero) through 9 (nine) and A (ten) through F (fifteen).

hierarchical tree structure. The organization of files on AIX, similar to tree-structure directories, with each file like a small branch of a larger branch that represents the file's parent directory. A directory can also

be contained in another higher level directory, with the parent of all directories represented by the tree's root (**root** or **root directory**).

highlight. To emphasize an area on the display by any of several methods, such as brightening the area or reversing the color of characters within the area.

history. A C-shell mechanism that lists previously executed commands. These commands can be re-executed with the ! command.

history file. A file containing a log of system actions and operator responses.

hog factor. In system accounting, an analysis of how many times each command was run, how much processor time and memory it used, and how intensive that use was.

hole. A block of binary zeros in a file.

home directory. (1) A directory associated with an individual user. (2) The user's current directory on login or after issuing the **cd** command with no argument.

home site. The computer that stores the modifiable copy of a user's home directory. This is the cluster site with the primary copy of his home directory if it is replicated. A user typically logs in to the computer that is his home site.

I/O. See **input/output**.

ID. Identification.

IF expressions. Expressions within a procedure, used to test for a condition.

indirect block. A block containing pointers to other blocks. Indirect blocks can be single-indirect, double-indirect, or triple-indirect.

informational message. A message providing information to the operator, that does not require a response.

initial program load (IPL). The process of loading the system programs and preparing the system to run jobs. See **initialize**.

initialize. To set counters, switches, addresses, or contents of storage to 0 or other starting values at the beginning of, or at prescribed points

in, the operation of a computer routine.

inline input. See **here document**.

inode. The internal structure for managing files in the system. Inodes contain all of the information pertaining to the node, type, owner, and location of a file. A table of inodes is stored near the beginning of a file system.

i-number. A number specifying a particular inode on a file system.

inodetab. An AIX kernel parameter that establishes a table in memory for storing copies of inodes for all active files.

input. Data to be processed.

input device. Physical devices used to provide data to a computer.

input file. A file opened by a program so that the program can read from that file.

input list. A list of variables to which values are assigned from input data.

input redirection. The specification of an input source other than the standard one.

input-output file. A file opened for input and output use.

input-output device number. A value assigned to a device driver by the guest operating system or to the virtual device by the virtual resource manager. This number uniquely identifies the device regardless of whether it is real or virtual.

input/output (I/O). Pertaining to either input, output, or both between a computer and a device.

interactive processing. A processing method in which each system user action causes response from the program or the system. Contrast with **batch processing**.

interface. A shared boundary between two or more entities. An interface might be a hardware component to link two devices together or it might be a portion of storage or registers accessed by two or more computer programs.

interleave factor. Specification of the ratio between contiguous physical blocks (on a fixed-disk) and logically contiguous blocks (as in a file).

interrupt. (1) To temporarily stop a process. (2) In data communications, to take an action at a receiving station that causes the sending station to end a transmission. (3) A signal sent by an I/O device to the processor when an error has occurred or when assistance is needed to complete I/O. An interrupt usually suspends execution of the currently executing program.

IPL. See **initial program load**.

job. (1) A unit of work to be done by a system. (2) One or more related procedures or programs grouped into a procedure.

job queue. A list, on disk, of jobs waiting to be processed by the system.

justify. To print a document with even right and left margins.

K-byte. See **kilobyte**.

kernel. The memory-resident part of the AIX Operating System containing functions needed immediately and frequently. The kernel supervises the input and output, manages and controls the hardware, and schedules the user processes for execution.

kernel parameters. Variables that specify how the kernel allocates certain system resources.

key pad. A physical grouping of keys on a keyboard (for example, numeric key pad, and cursor key pad).

keyboard. An input device consisting of various keys allowing the user to input data, control cursor and pointer locations, and to control the dialog between the user and the display station

keylock feature. A security feature in which a lock and key can be used to restrict the use of the display station.

keyword. One of the predefined words of a programming language; a reserved word.

keyword argument. One type of variable assignment that can be made on the command line.

kill. An AIX Operating System command that stops a process or sends a signal to it.

kill character. The character that is used to delete a line of characters entered after the user's prompt.

kilobyte. 1024 bytes.

kprocs. An AIX kernel parameter establishing the maximum number of processes that the kernel can run simultaneously.

label. (1) The name in the disk or diskette volume table of contents that identifies a file. See also **file name**. (2) The field of an instruction that assigns a symbolic name to the location at which the instruction begins, or such a symbolic name.

left justify. See **left-adjust**.

left margin. The area on a page between the left paper edge and the leftmost character position on the page.

left-adjust. The process of aligning lines of text at the left margin or at a tab setting such that the leftmost character in the line or field is in the leftmost position. Contrast with **right-adjust**.

library. A collection of functions, calls, subroutines, or other data.

Licensed Program (LP). Software programs that remain the property of the manufacturer, for which customers pay a license fee.

line editor. An editor that modifies the contents of a file one line at a time.

linefeed. An ASCII character that causes an output device to move forward one line.

link. A connection between an inode and one or more file names associated with it.

literal. A symbol or a quantity in a source program that is itself data, rather than a reference to data.

load. (1) To move data or programs into storage. (2) To place a diskette into a diskette drive, or a magazine into a diskette magazine drive. (3) To insert paper into a printer.

loader. A program that reads run files into main storage, thus preparing them for execution.

local. Pertaining to a device directly connected to your system without the use of a communications line. Contrast with **remote**.

<LOCAL> alias. The **<LOCAL>** alias can translate into different strings on different cluster sites for different processes. When **<LOCAL>** is the first component of the destination name for a symbolic link, it is replaced with its alias string, normally /**machinename**.

local cluster site. The site on a cluster that the user is logged in to. The term **local** normally refers to a TCF cluster site.

<LOCAL> file system. The part of the root file system hierarchy comprising system directories and files (such as the **/etc/motd** "message of the day" file) defined uniquely on a particular computer in the cluster. These files are not replicated. The name of the **<LOCAL>** file system appears in response to the **site-l** command.

location transparency. Allows an object to change location without the user's or program's knowledge if that location is not part of the object's name. For example, **/u/joe/glossary** may have been a file on **eyore** last week, but it is a file on **pooh** this week. Joe may not need to know that the file was on either **eyore** or **pooh**. If, however, Joe wants to find out where the site is located, he may invoke the **where** command.

log. To record; for example, to log all messages on the system printer. A list of this type is called a log, such as an error log.

log in. To begin a session at a display station.

log in shell. The program, or command interpreter, started for a user at log in.

log off. To end a session at a display station.

log out. To end a session at a display station.

logical device. A file for conducting input or output with a physical device.

loop. A sequence of instructions performed repeatedly until an ending condition is reached.

LP. See **licensed program**.

magic number. A constant located at a predefined offset in a file, used to verify the format of the file. Magic numbers are chosen because they are unlikely to occur as a random pattern in normal output.

mailbox. An area designated for storage of mail messages directed to a specific system user.

main storage. The part of the processing unit where programs are run and data are manipulated.

maintenance system. A special version of the AIX Operating System which is loaded from diskette and used to perform system management tasks.

major device number. A system identification number for each device or type of device.

mapped files. Files on the fixed-disk that are accessed as if they were in memory.

mask. A pattern of characters that controls the keeping, deleting, or testing of portions of another pattern of characters.

matrix. An array arranged in rows and columns.

maxprocs. An AIX kernel parameter establishing the maximum number of processes that can be run simultaneously by a user.

memory. Storage on electronic chips. Examples of memory are random access memory, read only memory, or registers. See **storage**.

menu. A displayed list of items from which an operator can make a selection.

message. (1) A response from the system to inform the operator of a condition which may affect further processing of a current program. (2) Information sent from one user in a multi-user operating system to another.

minidisk. A logical division of a fixed disk.

minor device number. A number used to specify various types of information about a particular device, for example, to distinguish among several printers of the same type.

mode word. An inode field that describes the type and state of the inode.

modem. See **modulator-demodulator**.

modulation. Changing the frequency or size of one signal by using the frequency or size of another signal.

modulator-demodulator (modem). A device that converts data from the computer to a signal that can be transmitted on a communications line, and converts the signal received to data for the computer.

module. A discrete programming unit that usually performs a specific task or set of tasks. Modules are subroutines and calling programs that are assembled separately, then linked to make a complete program.

monitor mode. A console display mode in which an application program can directly access the display adapter without conflict with the standard virtual terminal output mechanism.

mount. To make a file system accessible.

mountab. An AIX kernel parameter establishing the maximum number of file systems that can be mounted simultaneously.

multiprogramming. The processing of two or more programs at the same time.

multivolume file. A diskette file occupying more than one diskette.

IBM AIX Network File System (NFS). A licensed program that allows you to share files with other computers in one or more networks that have a variety of machine types and operating systems. You can mount file systems located on network servers and use remote files as if they were on your work stations by creating file trees that are independent of the file systems.

nest. To incorporate a structure or structures of some kind into a structure of the same kind. For example, to nest one loop (the nested loop) within another loop (the nesting loop); to nest one subroutine (the nested subroutine) within another subroutine (the nesting subroutine).

network. A collection of products connected by communication lines for information exchange between locations.

new-line character. A control character that causes the print or display position to move to the first position on the next line.

null. Having no value, containing nothing.

null character (NUL). The character hex 00, used to represent the absence of a printed or displayed character.

numeric. Pertaining to any of the digits 0 through 9.

object code. Machine-executable instruction, usually generated by a compiler from source code written in a higher level language. For programs that must be linked, object code consists of relocatable machine code.

octal. A base eight numbering system consisting of the digits 0 (zero) through 7 (seven).

opaque data. XDR opaque data is data of a fixed size that is passed to another machine without being interpreted.

open. (1) To make a file available to a program for processing.

operating system. Software that controls the running of programs; in addition, an operating system may provide services such as resource allocation, scheduling, input/output control, and data management.

operation. A specific action (such as move, add, multiply, load) that the computer performs when requested.

operator. (1) A symbol representing an operation to be done. (2) A person entering commands to the system.

output. The result of processing data.

output devices. Physical devices used by a computer to present data to a user.

output file. A file that is opened by a program so that the program can write to that file.

output redirection. The specification of an output destination other than the standard one.

override. (1) A parameter or value that replaces a previous parameter or value. (2) To replace a parameter or value.

overwrite. To write output into a storage or file space that is already occupied by data.

owner. The user who has the highest level of access authority to a data object or action, as defined by the object or action.

pad. To fill unused positions in a field with dummy data, usually zeros or blanks.

page. A block of instructions, data, or both.

page space minidisk. The area on a fixed disk that temporarily stores instructions or data currently being run. See also **minidisk**.

pagination. The process of adjusting text to fit within margins and/or page boundaries.

paging. The action of transferring instructions, data, or both between real storage and external page storage.

parallel processing. The condition in which multiple tasks are being performed simultaneously within the same activity.

parameter. Information that the user supplies to a panel, command, or function.

parent. Pertaining to a secured resource, either a file or library, whose user list is shared with one or more other files or libraries. Contrast with **child**.

parent directory. The directory one level above the current directory.

partition. See **minidisk**.

password. A string of characters that, when entered along with a user identification, allows an operator to sign on to the system.

password security. A program product option that helps prevent the unauthorized use of a display station, by checking the password entered by each operator at sign-on.

path name. See **full path name** and **relative path name**.

pattern-matching character. Special characters such as * or ? that can be

used in search patterns. Some used in a file specification to match one or more characters. For example, placing a ? in a file specification means any character can be in that position. Pattern-matching characters are also called wildcards.

permission code. A three-digit octal code, or a nine-letter alphabetic code, indicating the access permissions for a file. The access permissions are read, write, and execute.

permission field. One of the three-character fields within the permissions column of a directory listing indicating the read, write, and run permissions for the file or directory owner, group, and all others.

phase. One of several stages file system checking and repair performed by the **fsck** command.

physical device. See **device**.

physical file. An indexed file containing data for which one or more alternative indexes have been created.

physical record. (1) A group of records recorded or processed as a unit. Same as **block**. (2) A unit of data moved into or out of the computer.

PID. See **process ID**.

pipe. To direct the data so that the output from one process becomes the input to another process.

pipeline. A direct, one-way connection between two or more processes.

pitch. A unit of width of typewriter type, based on the number of times a letter can be set in a linear inch. For example, 10-pitch type has 10 characters per inch.

platen. The support mechanism for paper on a printer, commonly cylindrical, against which printing mechanisms strike to produce an impression.

pointer. A logical connection between physical blocks.

port. (1) To make the programming changes necessary to allow a program that runs on one type of computer to run on another type of computer. (2) An access point for data input to or data output from a computer system. See **connector**.

position. The location of a character in a series, as in a record, a displayed message, or a computer printout.

print queue. A file containing a list of the names of files waiting to be printed.

printout. Information from the computer produced by a printer.

priority. The relative ranking of items. For example, a job with high priority in the job queue will be run before one with medium or low priority.

priority number. A number that establishes the relative priority of printer requests.

privileged user. The account with superuser authority.

problem determination. The process of identifying why the system is not working. Often this process identifies programs, equipment, data communications facilities, or user errors as the source of the problem.

problem determination procedure. A prescribed sequence of steps aimed at recovery from, or circumvention of, problem conditions.

procedure. See **shell procedure**.

process. (1) A sequence of actions required to produce a desired result. (2) An entity receiving a portion of the processor's time for executing a program. (3) An activity within the system begun by entering a command, running a shell program, or being started by another process.

process accounting. An analysis of the use each process makes of the processing unit, memory, and I/O resources.

process ID (PID). A unique number assigned to a process that is running.

process transparency. The ability to execute and control tasks on any site in the cluster, regardless of where the user is logged in. The same system calls and commands are used, no matter where the process is located. For example, a remote job is aborted the same way that a local job is abandoned.

profile. (1) A file containing customized settings for a system or user. (2) Data describing the significant features of a user, program, or device.

program. A file containing a set of instructions conforming to a particular programming language syntax.

prompt. A displayed request for information or operator action.

propagation time. The time necessary for a signal to travel from one point on a communications line to another.

qdaemon. The daemon process that maintains a list of outstanding jobs and sends them to the specified device at the appropriate time.

queue. A line or list formed of items waiting to be processed.

queued message. A message from the system that is added to a list of messages stored in a file for viewing by the user at a later time. This is in contrast to a message that is sent directly to the screen for the user to see immediately.

quit. A key, command, or action that tells the system to return to a previous state or stop a process.

quote. To mask the special meaning of certain characters; to cause them to be taken literally.

random access. An access mode in which records can be read from, written to, or removed from a file in any order.

raw interface. In I/O, an interface in which data is not manipulated by the kernel before it arrives at the device driver.

readonly. Pertaining to file system mounting, a condition that allows data to be read, but not modified.

recovery procedure. (1) An action performed by the operator when an error message appears on the display screen. Usually, this action permits the program to continue or permits the operator to run the next job. (2) The method of returning the system to the point where a major system error occurred and running the recent critical jobs again.

redirect. To divert data from a process to a file or device to which it would not normally go.

reference count. In an inode, a record of the total number of directory entries that refer to the inode.

relational expression. A logical statement describing the relationship

(such as greater than or equal) of two arithmetic expressions or data items.

relational operator. The reserved words or symbols used to express a relational condition or a relational expression.

relative address. An address specified relative to the address of a symbol. When a program is relocated, the addresses themselves will change, but the specification of relative addresses remains the same.

relative addressing. A means of addressing instructions and data areas by designating their locations relative to some symbol.

relative path name. The name of a directory or file expressed as a sequence of directories followed by a file name, beginning from the current directory.

remote. Pertaining to a system or device that is connected to your system through a communications line. Contrast with **local**.

remote cluster site. A site on the cluster that the user is not logged in to. The term **remote** normally refers to a TCF cluster site.

replicated root file system. The replicated root system is a file system with key common files and directories for basic system operation. Almost all system commands, programs and libraries are in the replicated root file system. Other user and system file systems (like the local file system) are mounted on top of directories in the replicated root file system.

reserved character. A character or symbol that has a special (non-literal) meaning unless quoted.

reserved word. A word that is defined in a programming language for a special purpose, and that must not appear as a user-declared identifier.

reset. To return a device or circuit to a clear state.

restore. To return to an original value or image. For example, to restore a library from diskette.

right justify. See **right-adjust**.

right margin. The area on a page between the last text character and the right upper edge.

right-adjust. To place or move an entry in a field so that the rightmost character of the field is in the rightmost position. Contrast with **left-adjust**.

root. Another name sometimes used for superuser.

root directory. The top level of a tree-structured directory system.

root file system. The basic AIX Operating System file system, which contains operating system files and onto which other file systems can be mounted. The root file system is the file system that contains the files that are run to start the system running.

routine. A set of statements in a program causing the system to perform an operation or a series of related operations.

run. To cause a program, utility, or other machine function to be performed.

run-time environment. A collection of subroutines and shell variables that provide commonly used functions and information for system components.

scratch file. A file, usually used as a work file, that exists until the program that uses it ends.

screen. See **display screen**.

scroll. To move information vertically or horizontally to bring into view information that is outside the display screen boundaries.

sector. (1) An area on a disk track or a diskette track reserved to record information. (2) The smallest amount of information that can be written to or read from a disk or diskette during a single read or write operation.

security. The protection of data, system operations, and devices from accidental or intentional ruin, damage, or exposure.

segment. A contiguous area of virtual storage allocated to a job or system task. A program segment can be run by itself, even if the whole program is not in main storage.

semantic transparency. Allow the same command to function identically from all cluster sites. It provides, for example, for the **grep** command to have the same options and give the same results no matter where it is invoked.

separator. A character used to separate parts of a command or file.

sequential access. An access method in which records are read from, written to, or removed from a file based on the logical order of the records in the file.

server. A computer or process that provides the data, services, or resources that can be accessed by another computer or process in a network.

session. A collection of related processes sharing the same controlling terminal.

session leader. A process whose process ID, process group ID and session ID value are equal. The session leader is the one who establishes the controlling terminal, if any, for the session. Every process group belongs to exactly one session.

session records. In the accounting system, a record of time connected and line usage for connected display stations, produced from log in and log out records.

shared printer. A printer that is used by more than one work station.

shell. See **shell program**.

shell procedure. A series of commands combined in a file that carry out a particular function when the file is run or when the file is specified as an argument to the sh command. Shell procedures are frequently called shell scripts.

shell program. A program that accepts and interprets commands for the operating system.

shell prompt. The character string on the command line indicating the system can accept a command (typically the $ character).

shell script. See **shell procedure**.

shell variables. Facilities of the shell program for assigning variable values to names.

signal. An event that interrupts the normal execution of a process.

single-shift control. In code page switching, a control code that shifts to another page for a single character; nonlocking shifts.

size field. In an inode, a field that indicates the size, in bytes, of the file associated with the inode.

sort. To rearrange some or all of a group of items based upon the contents or characteristics of those items.

source diskette. The diskette containing data to be copied, compared, restored, or backed up.

source program. A set of instructions written in a programming language, that must be translated to machine language compiled before the program can be run.

special character. A character other than an alphabetic or numeric character. For example; *, +, and % are special characters.

special file. Special files are used in the AIX system to provide an interface to input/output devices. There is at least one special file for each device connected to the computer. Contrast with **directory** and **file**. See also **block special file** and **character special file**.

spool files. Files used in the transmission of data among devices.

stand-alone work station. A work station that can be used to preform tasks independent of (without being connected to) other resources such as servers or host systems.

standard error. The place where many programs place error messages.

standard input. The primary source of data going into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command.

standard output. The primary destination of data coming from a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can be to a file or another command.

stanza. A group of lines in a file that together have a common function. Stanzas are usually separated by blank lines, and each stanza has a name.

statement. An instruction in a program or procedure.

status. (1) The current condition or state of a program or device. For example, the status of a printer. (2) The condition of the hardware or software, usually represented in a status code.

storage. (1) The location of saved information. (2) In contrast to memory, the saving of information on physical devices such as disk or tape. See **memory**.

storage device. A device for storing and/or retrieving data.

string. A linear sequence of entities such as characters or physical elements. Examples of strings are alphabetic string, binary element string, bit string, character string, search string, and symbol string.

su. See **superuser**.

subdirectory. A directory contained within another directory in the file system hierarchy.

subprogram. A program invoked by another program.

subroutine. (1) A sequenced set of statements that may be used in one or more computer programs and at one or more points in a computer program. (2) A routine that can be part of another routine.

subscript. An integer or variable whose value refers to a particular element in a table or an array.

subshell. An instance of the shell program started from an existing shell program.

substitution. A procedure used by a text editor like **ed** or **vi** to replace one specified string of characters with another. If a global substitution is made, all occurrences of the specified text pattern are replaced with the new one.

substring. A part of a character string.

subsystem. A secondary or subordinate system, usually capable of operating independently of, or synchronously with, a controlling system.

super block. The most critical part of the file system containing information about every allocation or deallocation of a block in the file system.

superuser. Super user authority; root permissions.

superuser (su). The user who can operate without the restrictions designed to prevent data loss or damage to the system (User ID 0).

superuser authority. The unrestricted ability to access and modify any part of the operating system associated with the user who manages the system. The authority obtained when one logs in as **root**.

symbolic link. Type of file that contains the path name to another file as a directory; it functions as a pointer to the other file or directory. See **link**.

synchronous. Occurring in a regular or predictable sequence.

synchronous transmission. In data communication, a method of transmission in which the sending and receiving of characters is controlled by timing signals. Contrast with **asynchronous transmission**.

system. The computer and its associated devices and programs.

system call. A request by an active process for a service by the system kernel.

system customization. A process of specifying the devices, programs, and users for a particular data processing system.

system date. The date assigned by the system user during setup and maintained by the system.

system dump. A copy of memory from all active programs (and their associated data) whenever an error stops the system. Contrast with **task dump**.

system management. The tasks involved in maintaining the system in good working order and modifying the system to meet changing requirements.

system parameters. See **kernel parameters**.

system primary site. The machine (cluster site) designated to hold the primary copy of the replicated root file system. When files are changed in the replicated root file system, in primary site for the cluster must be available.

system profile. A file containing the default values used in system operations.

system-replicated file system. One that contains files and directories accessed by many users regardless of the users' specific applications. These system files, programs and directories are replicated on different sites in a cluster.

system unit. The part of the system that contains the processing unit, the disk drives, and the diskette drives.

system user. A person who uses a computer system.

system network architecture (SNA). A set of rules for controlling the transfer of information in a data communication network.

target diskette. The diskette to be used to receive data from a source diskette.

task. A basic unit of work to be performed. Examples are a user task, a server task, and a processor task.

task dump. A copy of memory from a program that failed (and its associated data). Contrast with **system dump**.

TCF. Transparent Computing Facility. TCF is an operating system that automatically allows for data, process, name, location and semantic transparency. Process transparency is the ability to execute and control tasks on any cluster site, no matter where the user program is currently executing. A TCF LPP is required to obtain support.

template. In enhanced edit mode, a character buffer associated with the terminal.

terminal. An input/output device containing a keyboard and either a display device or a printer. Terminals usually are connected to a computer and allow a person to interact with the computer.

text. A type of data consisting of a set of linguistic characters (for example, alphabet, numbers, and symbols) and formatting controls.

text application. A program defined for the purpose of processing text data (for example, memos, reports, and letters).

text editing program. See **editor** and **text application**.

texttab. A kernel parameter establishing the size of the text table, in memory, that contains one entry each active shared program text segment.

trace. To record data that provides a history of events occurring in the system.

trace table. A storage area into which a record of the performance of computer program instructions is stored.

track. A circular path on the surface of a fixed disk or diskette on which information is magnetically recorded and from which recorded information is read.

transfer. To move data from one location to another in a computer system or between two or more systems.

transmission control characters. In data communication, special characters that are included in a message to control communication over a data link. For example, the sending station and the receiving station use transmission control characters to exchange information; the receiving station uses transmission control characters to indicate errors in data it receives.

transparency. The obscuring of machine boundaries in a distributed system. The AIX/370 system supports several kinds of transparency, including name, location, semantic, data, and process transparency.

trap. An unprogrammed, hardware-initiated jump to a specific address. Occurs as a result of an error or certain other conditions.

tree-structured directories. A method for connecting directories such that each directory is listed in another directory except for the root directory, which is at the top of the tree.

true color adapters. In the Graphics Support Library, color adapters in which the pixel color value drives the digital-to-analog converters without the level of indirection forced by the video lookup table (VLT). Contrast with **VLT-based adapters**.

truncate. To shorten a field or statement to a specified length, discarding the remainder.

TTY. Designates a terminal. On a system with more than one terminal, the TTY field of the process status displayed by the **ps** command indicates which terminal started the process.

typematic key. A key that repeats its function multiple times when held down.

typestyle. Characters of a given size, style and design.

UID. See **user number**.

UNIX link. A mechanism that lets you use the **ln** command to assign more than one name to a file. Both the new name and the file being linked to must be in the same file system. A file is deleted when all the UNIX links (including the first link-- the original name) have been removed. Synonym for **hard link**.

update. An improvement for some part of the system.

user. The name associated with an account.

user account. See **account**.

user ID. See **user number**.

user name. A name that uniquely identifies a user to the system.

user number (UID). (1) A unique number identifying an operator to the system. This string of characters limits the functions and information the operator is allowed to use. The UID can often be substituted in commands that take a user's name as an argument.

user profile. A file containing a description of user characteristics and defaults (for example, printer assignment, formats, group ID) to be conveyed to the system while the user is signed on.

user-replicated file system. One that contains files and directories accessed only by specific users or for particular applications. These user files and directories are replicated on different sites in a cluster.

utility. A service; in programming, a program that performs a common service function.

valid. (1) Allowed. (2) True, in conforming to an appropriate standard or authority.

value. (1) In Usability Services, information selected or typed into a pop-up. (2) A set of characters or a quantity associated with a parameter or name. (3) In programming, the contents of a storage location.

variable. A name used to represent a data item whose value can change while the program is running. Contrast with **constant**.

verify. To confirm the correctness of something.

version. Information in addition to an object's name that identifies different modification levels of the same logical object.

video lookup table (VLT). In the Graphic Support Library, a table of indexes that contains a value for each of the red, green, and blue digital-to-analog converters on the VLT-based color adapter that drives the color guns in the display table.

virtual device. A device that appears to the user as a separate entity but is actually a shared portion of a real device. For example, several virtual terminals may exist simultaneously, but only one is active at any given time.

virtual storage. Addressable space that appears to be real storage. From virtual storage, instructions and data are mapped into real storage locations.

virtual terminal. Any of several logical equivalents of a display station available at a single physical display station.

VLT. See **video lookup table**.

VLT-based adapter. A color display adapter in which the pixel color value serves as an index into a video lookup table (VLT). The actual color resulting from a particular pixel color value depends on the values loaded into the VLT. Contrast with **true color adapter**.

Volume ID (Vol ID). A series of characters recorded on the diskette used to identify the diskette to the user and to the system.

wildcard. See **pattern-matching characters**.

word. A contiguous series of 32 bits (4 bytes) in storage, addressable as a unit. The address of the first byte of a word is evenly divisible by four.

work file. A file used for temporary storage of data being processed.

work station. A device at which an individual may transmit information to, or receive information from, a computer for the purpose of performing a task, for example, a display station or printer. See **programmable work station** and **dependent work station**.

working directory. See **current directory**.

wrap around. Movement of the point of reference in a file from the end of one line to the beginning of the next, or from one end of a file to the other.

zombie. A terminated process whose entry remains in the process table in the kernel.

Special Characters

Numerics

A

B

C

D

E

F

G

H

L

M

N

T

U

X