AIX Operating System
for the PS/2 and System/370
Programming Tools and Interfaces
Version 1.2.1

Document Number SC23-2304-01

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

Programming Tools and Interfaces

Version 1.2.1

Document Number SC23-2304-01

Edition Notice

Third Edition (March 1991)

This edition applies to Version 1.2.1 of the IBM Advanced Interactive Executive for Personal System/2 (AIX PS/2), Program Number 5713-AEQ, and to Version 1.2.1 of the IBM Advanced Interactive Executive for System/370 (AIX/370), Program Number 5713-AFL, 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.

Portions of the code and documentation were developed at the 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.

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 Acknowledgments

Trademarks and Acknowledgments

The following trademarks apply to this book:

IBM is a registered trademark of International Business Machine Corporation.

AIX is a registered trademark of 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 International Business Machine Corporation.

Ined is a trademark of INTERACTIVE Systems Corporation

UNIX is a registered trademark of UNIX System Laboratories, Inc. i the USA and other countries.

The Remote Procedure Call interface was developed by Sun Microsystems Inc. RPC is a trademark of Sun Microsystems, Inc.

Remote Procedure Call Language and eXternal Date Representation wer developed by the Sun Microsystems, Inc.

About This Book

This book describes the programming tools and services available for writing application programs that run on the IBM Advanced Interactive Executive (AIX) for the Personal System/2 and System/370. This book provides an overview of the programming process and describes how to use the programming tools and interfaces within that process. This book includes information you need to be able to:

- Understand the structure of the syste
- Design output for the displa
- Use C language programming tool
- Use system calls and library function
- Install programs on the syste
- Use the trace and error logging facilitie
- Write messages

Subtopics

Who Should Read This Book

This book is written for programmers or application developers who want to write, develop, and debug application programs that run on the AIX Operating System for the PS/2 and System/370.

What You Should Know

The book uses the C programming language in many of the examples, and many of the tools work only with C language source files. Therefore, you should be familiar with the C programming language to get the most out of this book. However, programmers who use other high-level languages can also benefit from the information in this book. In addition to knowing C language, you should:

Have experience in writing application program

Be able to use the AIX Operating System to

- Enter commands
- Create and delete files
- Edit files
- Move around the file system.

How to Use This Book

This book is a reference manual for application developers who are writing and testing programs that run under the AIX Operating System. For many topics, step-by-step instructions are given, but keep in mind that the book is not a tutorial. You can use the table of contents and the index to locate particular topics you want to review.

Subtopics
Highlighting

Highlighting

This book observes the following highlighting conventions:

New terms introduced in the text are shown in ***boldface italic***.
AIX commands, options, parameters, names of keys, keywords directives, and actual file names are in **boldface** type.
Structures are in **UPPERCASE BOLDFACE**.
Variable information is in *italic* type.
Anything users type is in **monospace** type.
Anything appearing on a display screen that is referred to in paragraph of text is in **monospace** type.
Instructions set off from a paragraph are printed in **monospace** type.

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 support and refers to other AIX books that contain more detailed information.

AIX Library Guide, Glossary, and Master Index, SC23-2324, describes the publications in the AIX Operating System library and contains a glossary of terms used throughout the library. This book also includes a master index to the contents of each of the publications in the library.

AIX Messages Reference, SC23-2294, lists messages displayed by the AIX Operating System and explains how to respond to them.

AIX 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 Technical Reference, SC23-2300 (Vol. 1) and SC23-2301 (Vol. 2), describes the system calls and subroutines a programmer uses to write application programs. This book also provides information about the AIX Operating System file system, special files, miscellaneous files, and the writing of device drivers.

AIX VS FORTRAN Reference, SC23-2050, describes the FORTRAN programming language as implemented on AIX RT, AIX PS/2, and AIX/370. This book describes all of the standard features of VS FORTRAN as well as the enhanced functions and capabilities incorporated into IBM AIX VS FORTRAN.

AIX VS FORTRAN User's Guide, SC23-2049, shows how to develop and execute FORTRAN programs on AIX RT, AIX PS/2, and AIX/370. This book also explains how to compile and execute programs that contain sections of code written in the VS Pascal and C programming languages.

AIX VS Pascal Reference, SC23-2054, describes the VS Pascal programming language as implemented on the IBM PS/2 or RT with the AIX Operating System installed. This book describes all of the standard features of Pascal as well as the enhanced functions and capabilities incorporated into IBM AIX VS Pascal.

AIX VS Pascal User's Guide, SC23-2053, shows how to develop and

execute Pascal programs on the IBM PS/2 and RT using the AIX Operating System. This book also explains how to compile and execute programs that contain sections of code written in the VS FORTRAN and C programming languages.

AIX X-Windows Programmer's Reference, SC23-2118, describes the X-Windows licensed program and provides information on X-Windows library functions, FORTRAN subroutines, protocols, and extensions.

AIX X-Windows User's Guide, SC23-2017, describes the X-Windows licensed program and shows how to start, run, install, and customize this program.

AIX PS/2 General Information, GC23-2055, describes the AIX PS/2 Operating System's functions and capabilities and the product's position in the AIX family of products.

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.

AIX PS/2 Text Formatting Guide, SC23-2044, describes the text formatting utilities available on the PS/2 and shows how to format text with NROFF and TROFF. This book also shows how to use the **vi** editor to create, revise, and store files.

AIX/370 Administration Guide, SC23-2088, describes such administrative tasks as updating the file system, backing up files, and fine-tuning and monitoring the performance of the operating system.

AIX/370 Diagnosis Guide, SC23-2090, describes procedures and tools that can be used to define and categorize symptoms of problems that may occur during daily operation.

AIX/370 General Information, GC23-2062, describes the functions and capabilities of AIX/370 and its position in the AIX family of products.

AIX/370 Planning Guide, GC23-2065, describes the functions and capabilities of the AIX/370 Operating System and lists the requirements for all supported hardware and software. This book also includes information to assist with planning for installation and customization of the operating system.

Installing and Customizing the AIX PS/2 Operating System, SC23-2290, provides step-by-step instructions for installing the AIX PS/2 Operating System and related programs. This book also shows how to customize the operating system to suit the user's specific needs and

work environment.

Installing and Customizing the AIX/370 Operating System, SC23-2066, provides step-by-step instructions for installing the AIX/370 Operating System and related programs. This book also shows how to customize the operating system to suit the user's specific needs and work environment.

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.

Using the AIX Operating System, SC23-2291, shows the beginning user how to use AIX Operating System commands to do such basic tasks as log in and out of the system, display and print files, and set and change passwords. It includes information for intermediate to advanced users about how to use communication and networking facilities and write shell procedures.

Table of Contents

Figures

1.0 Chapter 1. Programming with AIX

Subtopics

1.1 CONTENTS

1.2 About This Chapter

This chapter describes the IBM AIX tools and services for developing application programs. In addition, it indicates where to get more information about these facilities, both in this book and in other AIX books.

1.3 Programming Tools

The AIX system has many tools to help develop a C language program. These tools provide help in the following programming areas:

Entering a program into the syste

Checking a progra

Compiling and linking a progra

Correcting errors in a progra

Filing and maintaining a program

Subtopics
1.3.1 Entering a Program
1.3.2 Checking a Program
1.3.3 Compiling and Linking a Program
1.3.4 Correcting Errors in a Program
1.3.5 Building and Maintaining a Program

1.3.1 Entering a Program

The system has a line editor to help enter a program into a file to be compiled. The editor is called **ed**. Refer to *Using the AIX Operating System* for instructions about how to use this editor.

In addition, the system has two full screen editors, **INed** and **vi**. These editors display a full screen of data and allow interactive editing of the file.

1.3.2 Checking a Program

The following programs help check the format of a program for consistency and accuracy:

lint Checks for syntax, data type and other programming and usage errors. Refer to "Checking C Programs" in topic 2.4 for information about using this program.

cflow Generates a flow diagram of a C language program. Refer to "Other C Programming Tools" in topic 2.5 for information about this program.

cxref Generates a cross reference listing for a C language program. Refer to "Other C Programming Tools" in topic 2.5 for information about this program.

cb Reformats a C language source program into a consistent, indented format. Refer to "Other C Programming Tools" in topic 2.5 for information about this program.

1.3.3 Compiling and Linking a Program

The **cc** command compiles and links C, VS Pascal, VS FORTRAN, and Assembler language programs with one command line entry. Refer to Chapter 2, "Compiling and Linking Programs" for information about using this program.

1.3.4 Correcting Errors in a Program

The symbolic debug program, **dbx**, helps find logic errors in C, Pascal, and FORTRAN language programs. Refer to Chapter 12, "Debugging Programs" for information about using this program.

In addition, string searching programs such as **grep**, **sed** and **awk** help locate and change character strings (such as parameter names and syntax problems) in program files. Refer to Chapter 15, "Finding and Changing Strings" for information about using these programs.

1.3.5 Building and Maintaining a Program

Two programs help control changes to a program and build the final program module. These programs are:

make A program that builds programs from several source modules. It compiles only those modules that have changed. Refer to "Building Programs with make" in topic 2.7 for information about using this program.

sccs A set of programs that maintain separate versions of a program without storing separate copies of each version. Refer to Chapter 14, "Maintaining Different Versions of a Program" in topic 14.0 for information about using this program.

1.4 Program Development in a Heterogeneous Environment

Subtopics
1.4.1 Building Programs for Both AIX/370 and AIX PS/2

1.4.1 Building Programs for Both AIX/370 and AIX PS/2

It is possible to develop an AIX/370 LPP and an AIX PS/2 LPP from common source. If the programs are written in a machine independent language such as C, VS Pascal or VS Fortran, AIX with the Transparent Computing Facility (TCF) provides a means to develop both AIX/370 and AIX PS/2 versions of the program concurrently. To do so, you will need to do the development in a TCF cluster consisting of at least one AIX/370 site and one AIX PS/2 site. You will probably also want to make use of the hidden directory mechanism and design or make slight changes to your makefiles to account for multiple compilations of the same source files.

When designing your build method (both source directory layout and makefiles) the important thing is to arrange it so that object files for different CPU types (System/370 and 80386) are placed in separate directories. AIX doesn't provide tools to completely automate the building of programs for multiple CPU types, but the following simple approach describes one possible way to organize the source to accomplish this task conveniently. Large applications will require this approach to be extended to include placing source into an entire directory hierarchy.

First, place the common files (i.e., those source files that are machine independent) and a makefile in one directory. Place source files specific to the AIX PS/2 version of your applications in a subdirectory named i386, and source files specific to the AIX/370 version in a subdirectory named i370.

Then, build the AIX PS/2 version of the program by entering a command such as:

```
cd i386; onsite i386 make -f ../makefile
```

and build the AIX/370 version with a command as:

```
cd i370; onsite i370 make -f ../makefile
```

The makefile will need to access the common source files using a pathname such as **../filename.c**, but the objects created will be placed into the i386 or i370 subdirectory, as appropriate.

Makefile differences between AIX/370 and AIX PS/2 can be achieved by making use of the shell programs **/bin/u370** and **/bin/i386**, which are programs which exit with the value 0 (meaning true) or the value 1 (meaning false) depending on whether you are running on AIX/370 or AIX/PS/2.

Hidden directories can be used to give both the AIX/370 and AIX PS/2 versions of files the same name and make it so that the AIX/370 program is run by users on AIX/370 systems and the AIX PS/2 program is run by users on AIX PS/2 system. For example, if your application should be known as **/usr/bin/newprog**, you could install your program into a hidden directory with a sequence of commands such as the following:

```
mkdir -h /usr/bin/newprog
cp i386/newprog /usr/bin/newprog@/i386
chfstore i386 /usr/bin/newprog@/i386
cp i370/newprog /usr/bin/newprog@/i370
chfstore i370 /usr/bin/newprog@/i370
```

This can also be done by using **/etc/install** with the **-H** and **-v** options.

1.5 Programming Interfaces

When writing an application program for AIX, use the following ***system services***:

Command

Library routine

System calls

These services are available from C, VS FORTRAN, and VS Pascal language programs.

Subtopics
1.5.1 Commands
1.5.2 Library Routines
1.5.3 System Calls

1.5.1 Commands

To include the functions of any of the commands in a program, use the **fork** and **exec** system calls to allow the command to run in a part of the system (called a ***process***) that is separate from the program. The **system** library routine also runs a command in a program, and the **popen** library routine uses shell filters. When using commands in a program, ensure that these commands are also available on all systems that will use the program. Refer to *AIX Operating System Commands Reference* for details about commands.

1.5.2 Library Routines

Routines from the system libraries handle many complex or repetitive programming situations so that you can concentrate programming efforts on the unique programming situations. Details of each library subroutine are in *AIX Operating System Technical Reference*.

Some of the libraries on the system are:

C library
A collection of input/output formatting routines, system call interface routines and other functions. This library includes the library **stdio**, which is the standard input/output system.

Run Time Services library
A collection of routines that help a program use the following system services:

- Configuration
- Messages
- Trace
- Error log.

Math library
A collection of mathematics functions.

Extended curses library
A collection of routines for writing programs that help control display screen input and output without regard to the type of terminal that the system uses.

See Chapter 7, "Using the Subroutine Libraries" in topic 7.0 for a summary of the functions available in some of the libraries. See Chapter 9, "Controlling the Terminal Screen" in topic 9.0 for a description of using the **Extended curses** library.

1.5.3 System Calls

System calls are the lowest level of interaction between a program and the AIX operating system. System calls are called like library routines, and it is usually unnecessary for a programmer to know whether a particular routine is a system call or a library routine. In many cases, system calls provide a lower level service, while library routines make use of system calls to provide higher level services to an application. See Chapter 8, "Using System Calls" in topic 8.0 for examples of how to use many of the system calls.

2.0 Chapter 2. Compiling and Linking Programs

Subtopics

2.1 CONTENTS

2.2 About This Chapter

This chapter discusses the following programming processes:

- Compiling the progra
- Checking C program
- Other C programming tool
- Processing assembler language routine
- Building the program using the **make** utility program.

This chapter does not contain complete information about any of the programs. For complete information refer to the reference book for the language compiler or to *AIX Operating System Commands Reference* (for the C language compiler).

2.3 Compiling A Program

A ***compiler*** is a program that reads program text from a file and changes the programming language statements in that file to a form that the system can understand. The following steps show how the system creates this final form of the program:

1. Includes additional files specified with the **#include** directive, and expands macros into programming language statements. This applies to the C programming language only.

2. Changes the programming language statements into object code (a form that the system can understand). This object code is stored in a file with a **.o** suffix. This form of the program cannot be executed.

3. Links the object code (using the **ld** command) into a program that the system can execute. If you do not specify differently, the executable program is in the file **a.out** in the current directory.

When you compile a program on an AIX/370 or AIX PS/2 system, it generates an object file for that particular system. You can use this resulting object file only on the type of system on which it was compiled. For programs consisting of more than one object file, all object files must be compiled on the same system type in order for them to link together. Refer to *Using the AIX Operating System* for additional information.

If the program is written in the C language, use the **cc** program to perform these steps. See "Using the cc Program" in topic 2.3.2 and *AIX Operating System Commands Reference* for information about this program. If the program is written in assembler language, see "Processing Assembler Language Routines" in topic 2.6.

Subtopics
2.3.1 Choosing a Compiler
2.3.2 Using the cc Program

2.3.1 Choosing a Compiler

AIX PS/2 supports two C compilers. The C compiler released with AIX PS/2 Version 1.2 and earlier is known as the VS C compiler. The **vs** command invokes this compiler, which can be used to compile C language source code. The other compiler available on AIX PS/2 is called the Extended C compiler.

By default, if both AIX PS/2 C compilers are installed on a system, the **cc** command invokes the Extended C compiler. Only if the environment variable known as **COMPILER** is set to VSC does **cc** invoke the VS C compiler. If there is only one C compiler installed on the system, the **cc** command invokes that C compiler. The **vs** command always invokes the VS C compiler, if it is installed.

AIX/370 supports a compiler, which is also called the Extended C compiler. When you compile a program using this compiler, it generates an object file for AIX/370 only.

For information about invoking the compiler and the command-line options available for the **cc** command, see *C Language User's Guide*.

The following list includes some of the programming languages that are available for use with the AIX Operating System:

VS FORTRA
VS Pasca

Assembler

The books that come with the compiler programs contain information for using those languages. The examples in this chapter use the C language.

You can also write parts of the program in different languages and have one main routine call the separate routines to execute. To do this, however, follow the rules explained in the appropriate sections of the C, VS Pascal and VS FORTRAN Users Guides.

2.3.2 Using the cc Program

The **cc** program calls the C language compiler, but it can do much more. The **cc** program can:

Process the input with a macro preprocesso

Compile a high-level language progra

Assemble an assembly language progra

Link program modules

You can select any or all of these functions. In addition, you can replace the supplied programs for any of these steps with a program suited to special needs. *AIX Operating System Commands Reference* contains detailed reference information about the **cc** program.

On the System/370 you can generate load modules in either 370 mode (24-bit addressing) or in XA mode (31-bit addressing). You can use the XA mode for very large programs. The default is 370 mode which runs on either 370 machine type.

Subtopics
2.3.2.1 Examples of Commands

2.3.2.1 Examples of Commands

The following examples show some operations with the **cc** program using command line flags.

Compile source file **testfile.c** using the C library (**libc.a**). Link the resulting module and place the output in **a.out**:

```
cc testfile.c
```

Process source file **testfile.c** to produce assembler language output, and place the output in **testfile.s**:

```
cc testfile.c -S
```

Compile source file **testfile.c** using the C library (**libc.a**) and place the unlinked output in **testfile.o**:

```
cc testfile.c -c
```

Process source file **testfile.c** using the macro preprocessor only, and place the output in **testfile.i**:

```
cc testfile.c -P
```

Compile source file **testfile.c** using the C library (**libc.a**) but using the **newcpp** compiler program in the directory **/u/jim**. Place the unlinked output in **testfile.o**:

```
cc testfile.c -c -B/u/jim/new
```

2.4 Checking C Programs

Use the **lint** program to ensure that C programs do not contain syntax errors, as well as to verify that the programs do not contain data type errors. The **lint** program checks these areas of a program more carefully than the C compiler does, and displays many messages that point out possible problems. These messages may not require you to change the program if you decide to ignore the possible problems.

Refer to the *AIX Operating System Commands Reference* for detailed information on using the **lint** command.

The parameters for the **lint** command are in the following categories:

flags Optional flags to control **lint** messages. This section contains examples of some useful flags. See *AIX Operating System Commands Reference* for a complete list of the flags for the system.

file name The name of the C language source file for **lint** to check. The file name must end with **.c**.

library-name The name of a library that **lint** uses when checking the program. The following libraries are included with the system:

llib-lc.ln Checks standard function call syntax (included by default).

llib-port.ln Checks portable function call syntax (included by using the **-p** flag).

llib-lm.ln Checks math library call syntax (included by using the **-lm** flag).

llib-lcurses.l Checks extended curses library call syntax.

You can also create your own lint library. See "Creating A lint Library" in topic 2.4.8 for more information.

With no flags specified on the command line, the **lint** program checks the C source files and writes messages about the following coding errors and programming style differences that it finds:

Data types that are not used correctl

Variables and functions that are not use

Functions that are not used correctl

Syntax error

Techniques that could cause problems in moving the program to othe systems.

Subtopics
2.4.1 Operation
2.4.2 Program Flow
2.4.3 Data Type Checking
2.4.4 Variable and Function Checking

2.4.1 Operation

The **lint** program checks a group of files using the following procedure:

1. Checks each file and writes messages for problems found in that file.

2. Collects errors in included files and writes those messages.

3. Checks for consistency of labels and data types among the group of files.

4. Writes the source file name followed by a **?** (question mark) if any errors remain that are not assigned to either a source file or an included file.

If **lint** does not report any errors, the program has correct syntax and will compile without errors. Passing that test, however, does not mean that the program will operate correctly, or that the logic design of the program is accurate. The **lint** program does not check for design problems. It only checks language semantics and syntax.

2.4.2 Program Flow

The **lint** program detects parts of the program that cannot be reached. It writes messages about statements that do not have a label, but immediately follow statements that change the program flow, such as:

```
got
brea
continu
return
```

The **lint** program also detects and writes messages for the following conditions:

```
A loop that cannot be exited at the botto
A loop that cannot be entered at the to
Infinite loops such as
-   while(1)
-   for(;;)
```

Some programs that work may have such loops. However, the loops can cause problems.

The **lint** program does not detect functions that are called, but never return to the calling program. For example, a call to **exit** may result in code that cannot be reached, but **lint** does not detect it.

Programs generated by **yacc** and **lex** may have hundreds of **break** statements that cannot be reached. The **lint** program normally writes an error message for each of these **break** statements. Use the **-O** flag for the **cc** command when compiling the program to eliminate the resulting object code inefficiency, so that these extra statements are not important. Use the **-b** flag with the **lint** program to prevent writing of these messages when checking **yacc** and **lex** output code.

2.4.3 Data Type Checking

The **lint** program enforces the type checking rules of C language more strictly than the compiler does. In addition to the checks that the compiler makes, **lint** checks for the data type errors in the following areas:

Binary operators and implied assignment

Structures and union

Function definition and use

Enumerator

Type checking contro

Type casts

Subtopics
2.4.3.1 Binary Operators and Implied Assignments
2.4.3.2 Structures and Unions
2.4.3.3 Function Definition and Uses
2.4.3.4 Enumerators
2.4.3.5 Type Checking Control
2.4.3.6 Type Casts

2.4.3.1 Binary Operators and Implied Assignments

The C language allows mixing of the following data types in statements, and the compiler does not indicate an error when they are mixed:

```
cha
shor
in
lon
unsigne
floa
double
```

The language converts data types within this group automatically to allow the programmer more flexibility in programming. This flexibility, however, means that the programmer, **not** the language, must ensure that the data type mixing produces the desired result.

You can mix these data types when using them in the following ways (in the examples, **alpha** is type **char**, and **num** is type **int**):

Operands on both sides of an assignment operator, for example

```
alpha = num;
```

Operands in a conditional expression, for example

```
value = ( alpha < num ) ? alpha : num;
```

Operands on both sides of a relational operator, for example

```
if( alpha != num )
```

The type of an argument in a **return** statement is converted to the type of the value that the function returns. For example:

```
funct(x)        /* returns an integer */
{
  return( alpha );
}
```

The data types of pointers must agree exactly, except that you can mix arrays of **x**'s with pointers to **x**'s.

2.4.3.2 Structures and Unions

The **lint** program checks structure operations for the following requirements:

The left operand of the **->** operator must be a pointer to a structure.

The left operand of the **.** operator must be a structure.

The right operand of these operators must be a member of the sam structure.

The **lint** program makes similar checks for references to unions.

2.4.3.3 Function Definition and Uses

The **lint** program applies strict rules to function argument and return value matching. Arguments and return values must agree in type with the following exceptions:

You can match arguments of type **float** with arguments of type **double**.

You can match arguments within the following types

- char
- short
- int
- unsigned.

You can match pointers with the associated arrays

2.4.3.4 Enumerators

The **lint** program checks enumerated data type variables to ensure that:

Enumerator variables or members are not mixed with other types o other enumerators

The enumerated data type variables are only used in the followin areas:

- Assignment (=)
- Initialization
- Equivalence (==)
- Not equivalence (!=)
- Function arguments
- Return values.

2.4.3.5 Type Checking Control

To turn off strict type checking for one expression in the program, add the directive:

```
/*NOSTRICT*/
```

to the program immediately before the expression. This directive prevents strict type checking for only the next line in the program.

2.4.3.6 Type Casts

Type casts in the C language allows the program to treat data of one type as if it were data of another type. The **lint** program can check for type casts and write a message if it finds one.

The **-c** flag for the **lint** program controls the writing of comments about casts. Without the **-c** flag, **lint** treats casts as though they were assignments subject to messages. The resulting messages indicate the casts that are in the program. With the **-c** flag, **lint** ignores all legal casts.

2.4.4 Variable and Function Checking

The **lint** program detects variables and functions declared in the program, but not used. When it finds one of these cases, it writes a message. Variable and function errors that **lint** finds include the following:

Functions that return values inconsistentl

Variables and functions that are defined, but not use

Arguments to a function call that are not use

Functions that can return either with or without value

Functions that return values that are never use

Programs that use the value of a function when the function does no return a value.

Subtopics
2.4.4.1 Inconsistent Function Return
2.4.4.2 Function Values That Are Not Used
2.4.4.3 Disabling Function Related Error Messages

2.4.4.1 Inconsistent Function Return

If a function returns a value under one set of conditions, but does not return a value under another set of conditions, you cannot predict the results of the program. The **lint** program detects this type of error. For example, if both of the following statements are in a function definition:

```
return( expr );
```

and

```
return;
```

The **lint** program writes the message:

```
function name contains return(e); and return;
```

When using this function, the program may or may not receive a return value. The error message points out that problem.

The **lint** program also detects function returns caused by reaching the end of the function code (an implied return). For example, in the following part of a function:

```
checkout (a)
{
        if (a) return (3);
        fix_it ();
}
```

If **a** tests false, **checkout** calls **fix_it** and then returns with no defined return value. In this case, **lint** writes the message:

```
function checkout contains return(e); and return;
```

If **fix_it**, exits without returning, **lint** still writes the message even though nothing is wrong.

2.4.4.2 Function Values That Are Not Used

The **lint** program detects cases where a function returns a value and the calling program may not use the value. If the value is never used, the function definition may be inefficient and should be checked. If the value is sometimes used, the function may be returning an error code that the calling program does not check.

2.4.4.3 Disabling Function Related Error Messages

To prevent **lint** from reporting these types of errors, specify one or more of the following flags to the **lint** command:

-x Do not write messages about variables that are declared in an **extern** statement, but are never used.

-v Do not write messages about arguments to functions that are not used (except those that are also declared as register arguments).

-u Do not write messages about functions and external variables that are either used and not defined, or defined and not used. Use this flag to run **lint** on a subset of files of a larger program.

To prevent **lint** from reporting errors about unused arguments for one function, add the directive:

```
/*ARGSUSED*/
```

to the program before the function.

Add the following directive before the function definition to prevent the program from writing messages about variable numbers of arguments in calls to a function:

```
/*VARARGS*/
```

To check the first several arguments and leave the later arguments unchecked, add a digit to the end of the **VARARGS** directive to give the number of arguments that should be checked, such as:

```
/*VARARGS2*/
```

When **lint** reads this directive, it checks only the first two arguments.

When using **lint** with some (but not all) files that operate together, many of the functions and variables defined in those files may not be used. Also, many functions and variables defined elsewhere may be used. Use the **-u** flag to prevent **lint** from writing these messages.

2.4.5 Using Variables Before They Are Initialized

The **lint** program detects if a program uses a local variable (automatic and register storage classes) before assigning a value to it. In this case, using a variable also includes taking the address of the variable. This is because the program can use the variable (through its address) any time after it knows the address of the variable. Therefore, if the program does not assign a value to the variable before it finds the address of the variable, **lint** reports an error. Because **lint** only checks the physical order of the variables and their usage in the file, it may write messages about a program that actually does not contain errors.

The **lint** program recognizes and writes messages about:

Initialized automatic variable
Variables that are used in the expression that first sets the
Local variables that are set and never used

Note: The operating system initializes static and external variables to zero. Therefore, **lint** assumes that these variables are set (to zero) at the start of the program, and does not check to see if they have been assigned a value when they are used. When developing a program for a system that does not do this initialization, ensure that the program sets static and external variables to an initial value.

2.4.6 Portability Checking

Use **lint** to help ensure that you can compile and run the program on other systems that have a C language compiler that conforms to the UNIX System V requirements for a C compiler. The following paragraphs indicate areas to check before compiling the program on another system. Checking only these areas, however, does not guarantee that the program will run on any system.

Subtopics

2.4.6.1 Character Uses

Some systems define characters in a C language program as signed quantities with a range from -128 to 127; other systems define characters as positive values. The **lint** program writes messages when it finds character comparisons or assignments. The messages indicate that the use of characters may not be portable to other systems. For example, the fragment:

```
char c;
.

.

.

if( ( c = getchar() ) <0 )...
```

may work on one system but fail on systems like AIX where characters always take on positive values. The **lint** program writes the message:

```
nonportable character comparison
```

when it checks the program.

To make the program work on systems that use positive values for characters, declare **c** as an integer because **getchar** returns integer values.

2.4.6.2 Bit Field Uses

Bit fields may also produce problems when transferring a program to another system. When assigning constant values to bit fields, the field may be too small to hold the value, because bit fields may be signed quantities on the new system. To make this assignment work on all systems, declare the bit field to be of type **unsigned** before assigning values to it.

2.4.6.3 External Name Size

When changing from one type of system to another, be aware of differences in the information retained about external names during the loading process. The number of characters allowed for external names can vary. The AIX Operating System C language compiler considers at least the first 64 characters in internal and external identifiers as significant. Some programs that the compiler command calls and some of the functions that your programs call may further limit the number of significant characters in identifiers. In addition, the compiler keeps uppercase and lowercase characters separate. On other systems, uppercase or lowercase may not be important or allowed. To avoid problems with loading the program when transferring from one system to another:

1. Find out the requirements of each system.

2. Run **lint** with the **-p** flag.

The **-p** flag tells **lint** to change all external symbols to one case and limit them to six characters while checking the input files. The messages produced indicate the terms that may need to be changed.

2.4.6.4 Multiple Uses and Side Effects

Be careful when using complicated expressions. Many C compilers evaluate complex expressions in different orders. Function calls that are arguments of other functions may or may not be treated the same as ordinary arguments. Also, operators such as assignment, increment, and decrement may cause problems when used on another system. For example, if any variable is changed by a side effect of one of the operators and is also used elsewhere in the same expression, the result is undefined. The evaluation of the variable **years** in the following example is confusing because on some machines **years** is incremented before the function call, while on other machines **years** is incremented after the function call:

```
printf( "%d %d\n", ++years, amort( interest, years ) );
```

The **lint** program checks for simple scalar variables that may be affected by evaluation order problems. For example, the statement:

```
a[i]=b[i++];
```

causes **lint** to write the message:

```
warning: i evaluation order undefined
```

2.4.7 Coding Errors and Style Differences

Use **lint** to detect some coding errors and differences in coding style from the style that **lint** expects. Although coding style is mainly a matter of individual taste, examine each difference to ensure that the difference is both needed and accurate. The following paragraphs indicate the types of coding and style problems that **lint** can find.

Subtopics
2.4.7.1 Assignments of Long Variables to Integer Variables
2.4.7.2 Operator Precedence
2.4.7.3 Conflicting Declarations

2.4.7.1 Assignments of Long Variables to Integer Variables

If you assign variables of type **long** to variables of type **int**, the program may not work properly. The long variable is truncated to fit in the integer space and data may be lost. An error of this type occurs frequently when converting a program that uses **typedef** to run on a different system. When changing a **typedef** variable from **int** to **long**, the program can stop working because an intermediate result may be assigned to an integer variable, and the intermediate result is truncated.

To assign a long variable to an integer variable and prevent **lint** from writing messages for these assignments, use the **-a** flag with the **lint** program.

2.4.7.2 Operator Precedence

The **lint** program detects errors in operator precedence. Without parentheses to show order in complex sequences, these errors are hard to find by looking at the code. For example, the following statements are not clear:

```
if(x&077==0)...      /* actually: if(x & (077 == 0) ) */
                    /* should be: if( (x & 077) == 0) */
```

or

```
x<<2+40                 /* shift x left 42 positions */
                        /* should be: (x<<2) + 40 */
```

Use parentheses to make the operation more clearly understood. If you do not, **lint** writes a message.

2.4.7.3 Conflicting Declarations

The **lint** program writes messages about variables that are declared in inner blocks in a way that conflicts with their use in outer blocks. This practice is allowed but may cause problems in the program. Use the **-h** flag with the **lint** program to prevent writing of messages about conflicting declarations.

2.4.8 Creating A lint Library

For programming projects that define additional library routines, create an additional lint library to check the syntax of the programs. Using this library, the **lint** program can check the new functions in addition to the standard C language functions. Perform the following steps to create a new lint library (see the following paragraphs for more information about these steps).

1. Create an input file that defines the new functions.

2. Process the input file to create the lint library file.

3. Run **lint** using the new library.

Subtopics
2.4.8.1 Creating the Input File
2.4.8.2 Creating the lint Library File
2.4.8.3 Checking a Program with the New Library

2.4.8.1 Creating the Input File

Figure 2-1 shows an input file that defines three additional functions for **lint** to check. This file is a text file that you create with an editor. It consists of:

A directive to tell the **cpp** program that the following information is to be made into a library of **lint** definitions:

```
/*LINTLIBRARY*/
```

A series of function definitions that define

- The type of the function (**int** in the example)
- The name of the function
- The parameters that the function expects
- The types of the parameters
- A placeholder for any value that the function returns.

Name this file in the following format:

```
llib-l pgm
```

In this format, the letters **pgm** represent a unique name that indicates the functions contained in the input file. For example, in the example input file the name of this input file could be **llib-ldms**. When choosing the name of the file, ensure that it is not the same as any of the existing files in the **/usr/lib** directory.

```
/*LINTLIBRARY*/

#include <dms.h>


int  dmsadd( rmsdes, recbuf, reclen )
               int rmsdes;
               char *recbuf;
               unsigned reclen;
             { return 0; }
int  dmsclos( rmsdes)
               int rmsdes;
             { return 0; }
int  dmscrea( path, mode, recfm, reclen )
               char *path;
               int mode;
               int recfm;
               unsigned reclen;
             { return 0; }
```

Figure 2-1. Example lint Library Input File

2.4.8.2 Creating the lint Library File

To create a lint library file, process the input file using the following command:

```
/lib/cpp -C -Dlint llib-lpgm | /usr/lib/lint1 -Htmpfile > \
  /usr/lib/llib-lpgm.ln
```

This command tells the preprocessor program **cpp** and an intermediate program **lint1** to create a lint library file, **/usr/lib/llib-lpgm.ln** using the input file **llib-lpgm**. In each of these cases, the **pgm** in the file name represents the identifier for the input file. The file name **tmpfile** can be any temporary file name. The **lint1** program creates this file and uses it for intermediate storage. When the program completes, delete this file:

```
rm tmpfile
```

2.4.8.3 Checking a Program with the New Library

To check a program using the new library, use the command:

lint -l**pgm filename.c**

In this command, the letters *pgm* represent the identifier for the library, and *filename.c* represents the name of the file containing the C language source code to check. With no other flags, the **lint** program checks the C language source code against the standard lint library in addition to checking the indicated special lint library.

2.5 Other C Programming Tools

The AIX Operating System provides tools to help format and check the structure of the C language program. These tools include:

cb ***c beautifier***: This program formats the C language source program into a form that uses indentation levels to show the structure of the program.

cflow ***c flow diagram generator***: This program produces an output diagram that shows the logic flow of the C language source program.

cxref ***c cross reference list***: This program produces a list of all external references for each module of the C language program, including where the reference is resolved (if it is).

2.6 Processing Assembler Language Routines

To use program modules written in assembler language for the PS/2 and 370, assemble the source code and link the resulting output with any other modules in the program. To perform these steps, either:

1. Use the **as** program on a site of the appropriate type to assemble the source code into an object module.

2. Use the **ld** program to link the object modules with the other object modules that form the program.

or

Use the **cc** program to both assemble and link the program.

Subtopics
2.6.1 Using the as Program
2.6.2 Using the ld Program
2.6.3 Using the cc Program

2.6.1 Using the as Program

The following command sequences show some uses of the **as** program to assemble an assembler language module into an object module:

Assemble source file **asmtest.s** and place the output in the default file, **asmtest.o**

```
as asmtest.s
```

Assemble source file **asmtest.s** and place the output in the file **myfile.o**.

```
as -o myfile.o asmtest.s
```

The following command sequence shows how to use the **as** program to generate an assembler listing:

Assemble source file **asmtest.s** and write the assembler listing to the screen.

```
as -l asmtest.s
```

Note: An assembler listing written to standard output using the **-l** option is available only on AIX PS/2.

Assemble source file **asmtest.s** and place the assembler listing in the file **myfile**.

```
as -lmyfile asmtest.s
```

For more information about the **as** command, see **as** in *AIX Operating System Commands Reference*.

2.6.2 Using the ld Program

After assembling the source program with the **as** program, use the **ld** program to link that object module with other object modules, or to prepare it to run on the system.

2.6.3 Using the cc Program

To use the **cc** program to process an assembly language file, the file name must end in **.s** to indicate that it is an assembler language source file. The following command sequences show some uses of the **cc** program to assemble an assembler language module into an object module, and link it with other object modules to form the program:

Assemble and link the file **asmtest.s** and place the resulting program in file **a.out**.

```
cc asmtest.s
```

Assemble the file **asmtest.s** and place the resulting unlinked object code in file **asmtest.o**.

```
cc asmtest.s  -c
```

Assemble the file **asmtest.s**, link it with object files **oldfile.o** and **otherfile.o**, and place the resulting program in file **a.out**.

```
cc asmtest.s oldfile.o otherfile.o
```

You can also use the **cc** program to generate an assembler listing.

On the PS/2 assemble the file **asmtest.s** and place the assembler listing in **asmtest.lst**.

```
cc -X asmtest.s
```

On AIX/370, assemble the file **asmtest.s** and place an annotated assembler listing in **asmtest.lst**.

```
cc -Hasm asmtest.s > asmtest.lst
```

2.7 Building Programs with make

The **make** program builds up-to-date versions of programs. It keeps track of the commands that are needed to create the files, and uses a list of files that must be current before the operations can be done. After changing any part of a program, enter the **make** command on the command line. The **make** program then creates only the files that are affected by the change, according to the rules in its rules file.

Using the **make** program to maintain programs, you can:

- Combine the instructions for creating a large program in a single fil
- Define macros to use within the **make** description file
- Define new flags to use with the **make** program
- Create any file to use with the operating system, including SCCS file
- Use shell commands to define the method of file creation, or use th **make** program to create many of the basic types of files
- Create librarie
- Include files from other programs when creating a file

The **make** program is most useful for medium-sized programming projects. It does not solve the problems of maintaining more than one source version and describing huge programs (see Chapter 14, "Maintaining Different Versions of a Program" in topic 14.0).

Subtopics

2.7.1 Operation

The **make** program uses the following sources of information:

A description file that you creat

File name

Time stamps of the files from the file syste

Rules in the **make** program that tell how to build many of the standard types of files.

The file containing the completed program is called a **target file**. The **make** program creates a target file using a step-by-step procedure:

1. Finds the name of the target file in the description file, or in the **make** command.
2. Ensures that the files on which the target file depends exist and are up-to-date.
3. Determines if the target file is up-to-date with the files it depends on.
4. If the target file or one of the parent files is out of date, creates the target file using one of the following:
   a. Commands from the description file
   b. Internal rules to create the file (if they apply)
   c. Default rules from the description file.

If all files in the procedure are up-to-date when running the **make** program, **make** displays a message to indicate that the file is up-to-date, and then stops. If some files have changed, **make** creates only those files that are out of date, and does not create files that are already current.

When the **make** program runs commands to create a target file, it replaces macros with their values, writes each command line, and then passes the command to a new copy of the shell.

2.7.2 Using the make Program

Start the **make** program from the directory that contains the description file for the file to create. The variable name *desc-file* represents the name of that description file. Then, enter the command:

```
make -f desc-file
```

on the command line. Enter macro definitions, flags, description file names, and target file names along with the **make** command on the command line as follows:

```
make [flags] [macrodefinitions] [targets]
```

The **make** program then examines the command line entries to determine what to do. First, it looks at all macro definitions on the command line (entries that are enclosed in quotes and have equal signs in them) and assigns values to them. If it finds a definition for a macro on the command line different from the definition for that macro in the description file, it chooses the command line definition for the macro.

Next, the **make** program looks at the flags. See *AIX Operating System Commands Reference* for a list of the flags that **make** recognizes.

The **make** program expects the remaining command line entries to be the names of target files to be created. The **make** program creates the target files in left to right order. Without a target file name, the **make** program creates the first target file named in the description file that does not begin with a period. With more than one description file specified, **make** searches the first description file for the name of the target file.

2.7.3 Description Files

The description file tells **make** how to build the target file, what files are involved, and what their relationships are to the other files in the procedure. The description file contains the following information:

Target file nam

Parent file names that make up the target fil

Commands that create the target file from the parent file

Definitions of macros in the description file

The **make** program determines what files to create to get an up-to-date copy of the target file by checking the dates of the parent files. If any parent file was changed more recently than the target file, **make** creates the files that are affected by the change, including the target file.

If you name the description file **makefile** or **Makefile**, and are working in the directory containing that description file, enter the command:

```
make
```

to bring the first target file and its parent files up-to-date, regardless of the number of files that were changed since the last time **make** created the target file. In most cases, the description file is easy to write and does not change often.

To keep many different description files in the same directory, name them differently. Then, enter the command:

```
make -f desc-file
```

substituting the name of the description file to use in place of the variable name *desc-file*.

Subtopics
2.7.3.1 Format of a Description File Entry
2.7.3.2 Using Commands in a Description File
2.7.3.3 Calling the make Program from a Description File
2.7.3.4 Preventing the make Program from Writing Commands
2.7.3.5 Prevent Stopping on Errors
2.7.3.6 Determining the Trigger
2.7.3.7 Example of a Description File
2.7.3.8 Making the Description File Simpler

2.7.3.1 Format of a Description File Entry

The general form of an entry is:

```
target1 [target2..]:[:] [parent1..][; commands]
[#..]
[(TAB) commands] [#...]
...
```

The items that are inside brackets are optional. Targets and parents are file names (strings of letters, numbers, periods, and slashes). **make** recognizes wildcard characters such as ***** (asterisk) and **?** (question mark). Each line in the description file that contains a target file name is called a dependency line. Lines that contain commands must begin with a tab character.

Note: Because **make** uses the dollar sign symbol (**$**) to designate a macro, do not use that symbol in file names of targets and parents, or in commands in the description file unless you are using a defined **make** macro.

Put comments in the description file by using a # (number sign) to begin the comment phrase. The **make** program ignores the # and all characters on the same line after the #. The **make** program also ignores blank lines.

If the line is not a comment line, you can enter lines that are longer than the line width of the input device. To continue a line on the next line, put a **** (backslash) at the end of the line that is to be continued.

2.7.3.2 Using Commands in a Description File

A command is any string of characters not including a # or a new line. A command can use a # if it is in quotes. Commands can appear either after a semicolon on a dependency line, or on lines beginning with a tab immediately following a dependency line.

When defining the command sequence for a particular target, specify either one command sequence for each target in the description file, or specify separate command sequences for special sets of dependencies. Do not do both.

To use one command sequence for every use of the target, use a single **:** (colon) following the target name on the dependency line. For example:

```
test:        dependency list1...;
        command list...
           .
           .
           .
test:        dependency list2...;
```

defines a target name, **test**, with a set of parent files, and a set of commands to create the file. The target name, **test**, can appear in other places in the description file with another dependency list, but that name cannot have another command list in the description file. When one of the files that **test** depends on changes, **make** runs the commands in that one command list to create the file, **test**.

To specify more than one set of commands to create a particular target file, enter more than one dependency definition. Each dependency line must have the target name, followed by **::** (two colons), a dependency list, and a command list that **make** uses if any of the files in the dependency list changes. For example:

```
test::      dependency list1...;
        command list1...

test::      dependency list2...;
        command list2...
```

defines two separate processes to create the target file, **test**. If any of the files in dependency list1 changes, **make** runs command list1; if any of the files in dependency list2 changes, **make** runs command list2. To avoid conflicts, a parent file cannot appear in both dependency list1 and dependency list2.

Note: Because **make** passes the commands from each command line to a new shell, be careful when using certain commands (for example, **cd** and shell control commands) that have meaning only within a single shell process. The **make** program forgets these results before running the commands on the next line.

To group commands together, use the **** (backslash) at the end of a command line. The **make** program continues that command line into the next line in the description file. The shell sends both of these lines to a single new shell.

2.7.3.3 Calling the make Program from a Description File

Nest calls to the **make** program within a **make** description file by including the **$(MAKE)** macro in one of the command lines in the file. If this macro is present, **make** calls another copy of **make** even if the **-n** flag (do not execute) is set. The **make** program passes the flags to the new copy of **make** through the **MAKEFLAGS** variable.

If the **-n** flag is set when the **$(MAKE)** macro is found, the new copy of **make** does not do any of its commands, except another **$(MAKE)** macro. Use this characteristic to test a set of description files that describe a program. Enter the command:

```
make -n
```

For additional information on the **make -n** command, refer to the *AIX Operating System Commands Reference*.

The **make** program does not do any of the operations, but it writes all of the steps needed to build the program, including output from lower level calls to **make**.

2.7.3.4 Preventing the make Program from Writing Commands

To prevent **make** from writing the commands while it runs, do any of the following:

Use the **-s** flag on the command line when using the **make** command.

Put the fake target name **.SILENT** on a dependency line by itself in the description file. Because **.SILENT** is not a real target file, it is called a ***fake target***.

Put an @ in the first character position after the tab of each line in the description file that **make** should not write.

2.7.3.5 Prevent Stopping on Errors

The **make** program normally stops if any program returns an error code that is not zero. Some programs return status that has no meaning.

To prevent **make** from stopping on errors, do any of the following:

Use the **-i** flag on the command line when using the **make** command.

Put the fake target name **.IGNORE** on a dependency line by itself in the description file. Because **.IGNORE** is not a real target file, it is called a ***fake target***.

Put a **-** (hyphen) in the first character position after the tab of each line in the description file where **make** should not stop on errors.

2.7.3.6 Determining the Trigger

As you build larger and larger software products, relationships among the files involved become increasingly complex. Occasionally you may find it difficult to determine exactly which file(s) have changed to trigger the remake of the program being made. Even more difficult to determine is exactly which routine(s) from a given library are being called by a certain object module. The **make** command can determine these relationships for you.

Running **make** with the **-T** option causes trigger information to be printed to standard output. This gives the reason or the **make** rule by which **make** was activated.

```
libc.a(printf.o)<-pgm
```

The response above, for instance, indicates that *printf.o* was updated more recently than *pgm* so *pgm* was remade.

2.7.3.7 Example of a Description File

For example, a program named **prog** is made by compiling and loading three C language files **x.c, y.c,** and **z.c** with the C library (**libc.a**) and the run time library (**librts.a**). The files **x.c** and **y.c** share some declarations in a file named **defs**. The file **z.c** does not share those declarations. A description file to create **prog** looks like:

```
# Make prog from 3 object files
prog:  x.o y.o z.o
# Use the cc program to make prog
    cc  x.o y.o z.o -o prog

# Make x.o from 2 other files
x.o:   x.c defs
# Use the cc program to make x.o
    cc  -c x.c

# Make y.o from 2 other files
y.o:   y.c defs
# Use the cc program to make y.o
    cc  -c y.c

# Make z.o from z.c
z.o:   z.c
# Use the cc program to make z.o
    cc  -c z.c
```

If this file is called **makefile**, just enter the command:

```
make
```

to make **prog** up-to-date after making changes to any of the four source files **.x.c, y.c, z.c** or **defs**.

2.7.3.8 Making the Description File Simpler

To make this file simpler, use the internal rules of the **make** program. Using file system naming conventions, **make** knows that there are three **.c** files corresponding to the needed **.o** files. It also knows how to generate an object from a source file (that is, issue a **cc -c** command).

By taking advantage of these internal rules, the description file becomes:

```
# Make prog from 3 object files
prog:  x.o y.o z.o
# Use the cc program to make prog
    cc  x.o y.o z.o -o prog

# Use the file defs and the .c file
# when making x.o and y.o
x.o y.o:   defs
```

2.7.4 Internal Rules

The internal rules for the **make** program are in a file that looks like a description file. With the **-r** flag the **make** program does not use the internal rules file; you must supply the rules to create the files. The internal rules file contains a list of file name suffixes (such as, **.o**, or **.a**) that **make** understands, plus the rules that tell **make** how to create a file with one suffix from a file with another suffix. If you do not change the list, **make** understands the following suffixes:

.o Object file

.c C source file

.e Efl source file

.r Ratfor source file

.f FORTRAN source file

.s Assembler source file

.y Yacc-c source grammar

.yr Yacc-Ratfor source grammar

.ye Yacc-Efl source grammar

.l Lex source grammar.

The list of suffixes is like a dependency list in a description file, and follows the fake target **.SUFFIXES**. Because **make** looks at the suffixes list in left to right order, the order of the entries is important. The **make** program uses the first entry in the list that satisfies two requirements:

The entry matches input and output suffix requirements

The entry has a rule assigned to it

The **make** program creates the name of the rule from the two suffixes of the files that the rule defines. For example, the name of the rule to transform a **.r** file to a **.o** file is **.r.o**.

To add more suffixes to the list, add an entry for **.SUFFIXES** in the description file. For a **.SUFFIXES** line without any suffixes following the target name in the description file, **make** erases the current list. To change the order of the names in the list, erase the current list and then assign a new set of values to **.SUFFIXES**.

Figure 2-2 shows the paths that **make** uses to create a file. If two paths connect a pair of suffixes, **make** uses the longer one only if the intermediate file exists or is named in the description file.

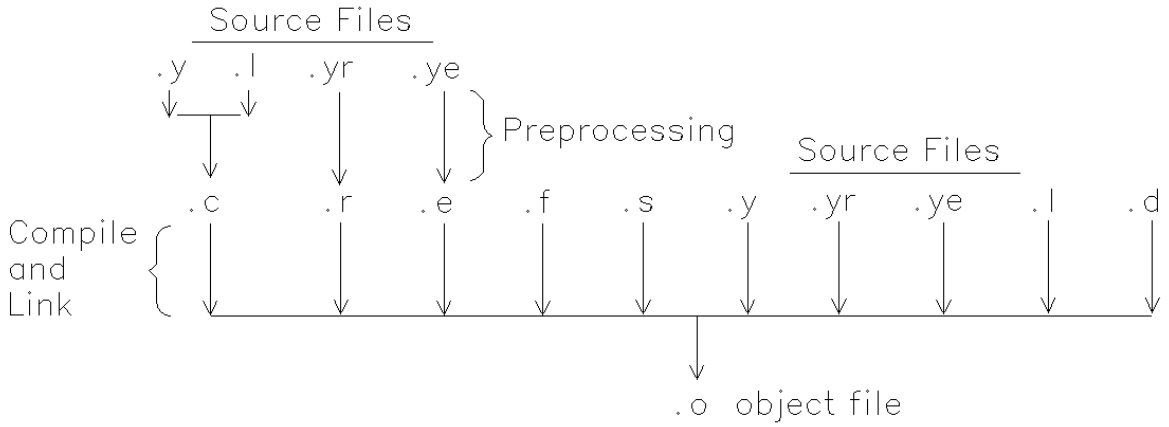


Figure 2-2. Rules for Creating Files

Subtopics
2.7.4.1 Example of Default Rules File
2.7.4.2 Single Suffix Rules
2.7.4.3 Using Make with Archive Libraries
2.7.4.4 Changing Macros in the Rules File

2.7.4.1 Example of Default Rules File

Figure 2-3 shows a portion of the default rules file.

```
  # Define suffixes that make knows
  .SUFFIXES:  .o .c .e .r .f .y .yr .ye .l .s

  # Begin macro definitions for
  # internal macros
  YACC = yacc
  YACCR = yacc -r
  YACCE = yacc -e
  YFLAGS =
  LEX = lex
  LFLAGS =
  CC =cc
  AS = as
  CFLAGS =
  RC = rc
  RFLAGS =
  EC = ec
  EFLAGS =
  FFLAGS =
  # End macro definitions for
  # internal macros

  # Create a .o file from a .c
  # file with the cc program
  .c.o:
           $(CC) $(CFLAGS) -c $<

  # Create a .o file from either a
  # .e , a .r , or a .f
  # file with the efl compiler
  .e.o .r.o .f.o:
           $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<


  # Create a .o file from
  # a .s file with the assembler
  .s.o:
           $(AS) -o $@ $<


  .y.o:
  # Use yacc to create an intermediate file
           $(YACC) $(YFLAGS) $<
  # Use cc compiler
           $(CC) $(CFLAGS) -c y.tab.c
  # Erase the intermediate file
           rm y.tab.c
  # Move to target file
           mv y.tab.o $@

  .y.c:
  # Use yacc to create an intermediate file
           $(YACC) $(YFLAGS) $<
  # Move to target file
```

```
        mv y.tab.c $@
```

Figure 2-3. Example Default Rules File

2.7.4.2 Single Suffix Rules

The **make** program also has a set of single suffix rules to create files directly to a file name that does not have a suffix (command files for example). The **make** program has rules to change the following source files with a suffix to object files without a suffix:

.c: From a C language source file

.c~: From an SCCS C language source file

.sh: From a shell file

.sh~: From an SCCS shell file.

Therefore, to maintain a program like **cat**, enter:

```
make cat
```

if all of the needed files are in the current directory.

2.7.4.3 Using Make with Archive Libraries

Use **make** to build libraries and library files. The **make** program recognizes the suffix **.a** as a library file. The internal rules for changing source files to library files are:

.c.a C source to archive

.c~.a SCCS C source to archive

.s~.a SCCS assembler source to archive.

2.7.4.4 Changing Macros in the Rules File

The **make** program uses macro definitions in the rules file. To change these macro definitions, enter new definitions for those macros on the command line or in the description file. The **make** program uses the following macro names to represent language processors that it uses:

AS for the Assemble
CC for the C compile
RC for the ratfor compile
EC for the efl compile
YACC for **yacc**
YACCR for **yacc -r**
YACCE for **yacc -e**
LEX for **lex**.

The **make** program uses the following macro names to represent flags that it uses:

CFLAGS for C compiler flag
RFLAGS for ratfor compiler flag
EFLAGS for efl compiler flag
YFLAGS for **yacc** flags
LFLAGS for **lex** flags.

Therefore, the command:

```
make "CC=newcc"
```

tells **make** to use the **newcc** program in place of the usual C language compiler. Similarly, the command:

```
make "CFLAGS=-O"
```

tells **make** to optimize the final object code produced by the C language compiler.

To look at the internal rules (in **rules.c**) that **make** uses, enter the following command:

```
make -p -f /dev/null 2>/dev/null
```

The output appears on the standard output.

2.7.5 Defining Default Conditions

When **make** creates a target file and cannot find commands in the description file and internal rules to create a file, it looks at the description file for default conditions. To define the commands that **make** performs in this case, use the **.DEFAULT** target name in the description file:

```
.DEFAULT:
        command
        command
            .
            .
            .
```

Because **.DEFAULT** is not a real target file, it is called a fake target. Use the **.DEFAULT** fake target for an error recovery routine, or for a general procedure to create all files in the program that are not defined by an internal rule of the **make** program.

2.7.6 Including Other Files

Include files other than the current description file by using the word **include** as the first word on any line in the description file. Follow the word with a blank or a tab, and then the set of file names for **make** to include in the operation. For example:

```
include    /u/tom/temp /u/tom/sample
```

tells **make** to read the files **temp**, **sample** and the current description file to build the target file.

If any of the files to be included are missing, **make** returns an error and exits. If this is not what you want, you can use an alternative keyword, **oinclude**. It uses the same format:

```
oinclude /u/tom/temp /u/tom/sample
```

If **oinclude** finds missing files, no error is reported and **make** continues as if the **oinclude** command had not been encountered.

Do not use more than 16 levels of nesting with the include files feature.

2.7.7 Defining Macros

A macro is a name (or label) to use in place of several other names. It is a shorthand way of using the longer string of characters. To define a macro:

1. Start a new line with the name of the macro.

2. Follow the name with an **=** (equal sign).

3. To the right of the **=**, enter the string of characters that the macro name represents.

The macro definition can contain blanks before and after the **=** without affecting the result. The macro definition cannot contain a **:** (colon) or a tab before the **=**.

The following are examples of macro definitions:

```
# Macro "2" has a value of "xyz"
2 = xyz

# Macro "abc" has a value of "-ll -ly"
abc = -ll -ly

# Macro "LIBES" has a null value
LIBES =
```

A macro that is named but is not defined has a value of the null string.

2.7.8 Using Macros in a Description File

After defining a macro in a description file, use the macro in the description file commands by putting a **$** (dollar sign) before the name of the macro. If the macro name is longer than one character, put **( )** (parentheses) or **{ }** (braces) around it. The following are examples of using macros:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two examples have the same effect.

The following fragment shows how to define and use some macros:

```
# OBJECTS is the 3 files x.o, y.o and
# z.o (previously compiled)
OBJECTS = x.o y.o z.o

# LIBES is the standard library
LIBES = -lc

# prog depends on x.o y.o and z.o
prog:  $(OBJECTS)
# Link and load the 3 files with
# the standard library to make prog
      cc $(OBJECTS) $(LIBES) -o prog
```

The **make** program that uses that description file loads the three object files with the **libc.a** library.

A macro definition entered on the command line replaces any macro definitions in the description file that define the same macro label. Therefore, the command:

```
make "LIBES= -ll"
```

loads the files with the Lex (**-ll**) library.

Note: When entering macros with blanks in them on the command line, put **"** (double quotes) around the macro. Without the double quotes, the shell interprets the blanks as parameter separators and not a part of the macro.

2.7.9 Internal Macros

The **make** program has built-in macro definitions for use in the description file. These macros help specify variables in the description file. The **make** program replaces the macros with one of the following values:

$@ The name of the current target file

$$@ The label name on the dependency line

$? The names of the files that have changed more recently than the target

$< The name of the out-of-date file that caused a target file to be created

$* The name of the current parent file without the suffix

$% The name of an archive library member.

Subtopics
2.7.9.1 Target File Name
2.7.9.2 Label Name
2.7.9.3 Younger Files
2.7.9.4 First Out-of-date File
2.7.9.5 Current File Name Prefix
2.7.9.6 Archive Library Member

2.7.9.1 Target File Name

If the **$@** macro is in the command sequence in the description file, **make** replaces the symbol with the full name of the current target file before passing the command to the shell to be run. The **make** program replaces the symbol only when it runs commands from the description file to create the target file.

2.7.9.2 Label Name

If the **$$@** macro is on the dependency line in a description file, **make** replaces this symbol with the label name that is on the left side of the colon in the dependency line. This name could be a target file name, the name of a new flag, or the name of another macro. For example, if the following is included in a dependency line:

```
cat:     $$@.c
```

The **make** program translates it to:

```
cat:     cat.c
```

when **make** evaluates the expression. Use this macro to build a group of files, each of which has only one source file. For example, to maintain a directory of system commands, use a description file like:

```
# Define macro CMDS as a series
# of command names
CMDS = cat dd echo date cc cmp comm ar ld chown

# Each command depends on a .c file
$(CMDS):      $$@.c
# Create the new command set by compiling the out of
# date files ($?) to the target file name ($@)
         $(CC) -O $? -o $@
```

The **make** program changes the **$$(@F)** macro to the file part of **$@** when it runs. For example, use this symbol when maintaining the **/usr/include** directory while using a description file in another directory. That description file would look like:

```
# Define directory name macro INCDIR
INCDIR = /usr/include

# Define a group of files in the directory
# with the macro name INCLUDES
INCLUDES = \
         $(INCDIR)/stdio.h \
         $(INCDIR)/pwd.h \
         $(INCDIR)/dir.h

# Each file in the list depends on a file
# of the same name in the current directory
$(INCLUDES):       $$(@F)
# Copy the younger files from the current
# directory to /usr/include
         cp $? $@
# Set the target files to read only status
         chmod 0444 $@
```

This description file creates a file in the **/usr/include** directory when the corresponding file in the current directory has been changed.

2.7.9.3 Younger Files

If the **$?** macro is in the command sequence in the description file, **make** replaces the symbol with a list of parent files that have been changed since the target file was last changed. The **make** program replaces the symbol only when it runs commands from the description file to create the target file.

2.7.9.4 First Out-of-date File

If the **$<** macro is in the command sequence in the description file, **make** replaces the symbol with the name of the file that started the file creation. The file name is the name of the parent file that was out of date with the target file, and therefore caused **make** to create the target file again.

In addition, use a letter (D or F) after the **<** (less-than sign) to get either the directory name (D) or the file name (F) of the first out-of-date file. For example, if the first out-of-date file were:

```
/u/tom/sample.c
```

then **make** gives the following values:

```
$(<D)   =   /u/tom
$(<F)   =   sample
$<      =   /u/tom/sample
```

The **make** program replaces this symbol only when it runs commands from its internal rules or from the .DEFAULT list.

2.7.9.5 Current File Name Prefix

If the **$*** macro is in the command sequence in the description file, **make** replaces the symbol with the file name part (without the suffix) of the parent file that **make** is currently using to generate the target file. For example, if **make** is using the file:

```
test.c
```

then the **$*** represents the file name, **test**.

In addition, use a letter (D or F) after the * (asterisk) to get either the directory name (D) or the file name (F) of the current file.

For example, **make** uses many files (specified either in the description file or the internal rules) to create a target file. Only one of those files (the ***current file***) is used at any moment. If that current file were:

```
/u/tom/sample.c
```

then **make** gives the following values for the macros:

```
$(*D)  =  /u/tom
$(*F)  =  sample
$*     =  /u/tom/sample
```

The **make** program replaces this symbol only when it runs commands from its internal rules (or from the .DEFAULT list), and not when running commands from a description file.

2.7.9.6 Archive Library Member

If the **$%** macro is in a description file, and the target file is an archive library member, **make** replaces the macro symbol with the name of the library member. For example, if the target file is:

```
lib(file.o)
```

then **make** replaces the **$%** with the member name, **file.o**.

2.7.10 Changing Macro Definitions in a Command

When macros in the shell commands are in the description file, you can change the values that **make** assigns to the macro. To change the assignment of the macro, put a **:** (colon) after the macro name, followed by a replacement string. The form is as follows:

```
$(macro:string1=string2)
```

When **make** reads the macro and begins to assign the values to the macro from the macro definition, it replaces each **string1** in the macro definition with a value of **string2**. For example, if the description file contains the macro definition:

```
FILES=test.o sample.o form.o defs
```

you can replace the file **form.o** with a new file, **input.o**, by using the macro in the description file commands:

```
cc -o $(FILES:form.o=input.o)
```

Changing the value of a macro in this manner is useful when maintaining archive libraries (see the **ar** program in *AIX Operating System Commands Reference*).

2.7.11 Using Make with SCCS Files

The **make** program does not allow references to prefixes of file names. Because SCCS file names begin with an **s.**, do not refer to them directly within a **make** description file. The **make** program uses a different suffix, the **~** (tilde), to represent SCCS files. Therefore, **.c~.o** refers to the rule that transforms an SCCS C language source file into an object. The internal rule is:

```
.c~.o:
        $(GET) $(GFLAGS) -p  $<  >$*.c
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c
```

The **~** added to any suffix changes the file search into an SCCS file name search with the actual suffix named by the . (period) and all characters up to (but not including) the **~**. The **GFLAGS** macro passes flags to SCCS to determine the version of the SCCS files to be used.

The **make** program recognizes the following SCCS suffixes:

.c~ C source

.y~ **yacc** source grammar

.s~ Assembler source

.sh~ shell

.h~ header

The **make** program has internal rules for changing the following SCCS files:

```
.c~:
.sh~:
.c~.o:
.s~.o:
.y~.o:
.l~.o:
.y~.c:
.c~.a:
.s~.a:
.h~.h:
```

Subtopics
2.7.11.1 Description Files Stored in SCCS

2.7.11.1 Description Files Stored in SCCS

If you specify a description file, or a file named **makefile** is in the current directory, **make** does not look for a description file within SCCS. If a description file is not in the current directory and you enter the command **make**, without specifying a description file the **make** program looks for an SCCS file named either **s.makefile** or **s.Makefile**. If either of these files are present, **make** uses a **get** command to tell SCCS to build the description file from that source file. The value of the internal macro, **GETFLAGS**, determines the level of the file that SCCS creates. When SCCS creates the description file, **make** uses the file as a normal description file. When **make** finishes, it removes the created description file from the current directory.

2.7.12 How make Uses the Environment Variables

Each time **make** runs, it reads the current environment variables and adds them to its defined macros. In addition, it creates a new macro called **MAKEFLAGS**. This macro is a collection of all input flags to the **make** program (without the minus signs). Command line flags and assignments in the description file can also change the **MAKEFLAGS** macro. When **make** starts another process, it passes **MAKEFLAGS** to that process by using the **export** command.

When **make** runs, it assigns macro definitions in the following order:

1. Reads the **MAKEFLAGS** environment variable to set debug on, if it is needed.

   If **MAKEFLAGS** is not present or null, **make** sets its internal **MAKEFLAGS** variable to the null string. Otherwise, **make** assumes that each letter in **MAKEFLAGS** is an input flag. The **make** program uses these flags (except for the **-f, -p,** and **-r** flags) to determine its operating conditions.

2. Reads and sets the input flags from the command line. The command line adds to the previous settings from the **MAKEFLAGS** environment variable.

3. Reads macro definitions from the command line. **Make** ignores any further assignments to these names.

4. Reads the internal macro definitions.

5. Reads the environment, including the **MAKEFLAGS** macro. The **make** program treats the environment variables as macro definitions and passes them to other shell programs.

2.7.13 Tracking Dependencies

make was created to keep track of medium-to-large size software projects in which the build process is often complex. When many modules depend on a single section of code, a small change can have very broad consequences. A single change to **/usr/include/stdio.h**, for instance, could cause hundreds of object modules in a single project to be remade. The programmers in charge of such projects need to be kept aware of these dependencies and AIX provides a series of interlocking tools for doing so.

make itself provides the **-n** and the **-T** options.

make -n lists all the commands used to build a given program, without actually running them. Since a single run of **make** can call another **make**, this can sometimes be an overwhelming volume of information.

make also provides the **-T** option, which displays just the dependency triggers for a given program. If the program needs to be remade, **make -T** displays the lowest level reason or **make** rule which would cause **make** to remake the file. If everything is up to date and the program does not need to be remade, **make -T** returns no output.

When **make** is used for preparing a C program, **cpp** and **ld** will be included in the build process. **cpp** is responsible for inserting **header** files in the finished program. **ld** is responsible for incorporating library routines.

Both of these programs offer options for dependency tracking. Some of these options produce output in a format suitable for use in **make** description files. When **make** runs **ld** and **cpp** with these options, the dependency information can be directed back into **make**'s own description files. In this way, running **make** today will automatically record any new dependencies created. Then future invocations of **make** will automatically take those new dependencies into account.

Subtopics

2.7.13.1 ld options for use in make

When you need to determine which archive members and object modules are used in creating a given target file, **ld** gives you a quick, convenient way to do this. **ld -m** writes to standard output the names of all the files and archive members being used in that link operation, along with a map of how they are arranged in memory. This output can be redirected to a file, yielding a record of the operation which can be examined afterward.

```
ld -m -oexecutable  file1.o  file2.o  mylib.a > logfile
```

This command creates *logfile. logfile* records the fact that *file1.o, file2.o,* and specified members of *mylib.a* were used in the creation of a program named *executable*. It writes the data in a table format, unsuitable for use in **make** description files.

The paired options, **ld -q** and **ld -Q**, allow you to specify the exact name of the description file into which the information is to be put and exactly how you want the file name to be reported on each dependency line in that file.

```
ld  -otarget file1.o file2.o libc.a -Qdesc-file
```

The command above will build *target* and record the dependency information in *desc-file*. In certain situations the name of *target* may not be reported in exactly the format needed by **make**. When this happens, you can add the **-q***target* option, which allows you to specify exactly how you want the name *target* reported.

```
ld  -otarget file1.o file2.o libc.a -qtarget -Qdesc-file
```

By using **-qtarget**, the format of the name "target" is exactly what is needed in a **make** description file. The format of the lines in *desc-file* is:

```
target:  dependency
```

dependency is determined by which library modules are referenced by *target* during the link operation. It has the form:

```
full-pathname/libname(member)
```

Notice that this output cites exactly which library routine is being called. If *target* references *printf.o* in *libc.a*, the line would look like this:

```
target:  /lib/libc.a(printf.o)
```

desc-file is often used to guide further invocations of **make**. Thus the dependency lines being inserted must be added to the file rather than overwriting it. It is not created anew each time. **ld** preens the file of any prior entries for *target* and the new dependency lines are added. Then it is sorted with the **-u** option, which eliminates duplicate entries.

2.7.13.2 cpp options for use in make

The **cpp** preprocessor offers dependency tracking facilities similar to those of **ld**.

cpp -M generates **makefile** dependencies and sends the results to standard error. It automatically converts the *filename.c* given on the **cc** command line to *filename.o filename.o* is automatically the target file name for which dependencies are reported. This is not a format suitable for use in **make** description files.

If you wish dependency information in a format suited to use by **make**, this can be obtained with two **cpp** options that function parallel to the **ld** options described above. These are the paired **-t** and **-X** options.

cpp -t*target* allows you to specify the actual name to be reported on the dependency line. **cpp -X***desc-file* gives the name of the file to which the information is to be written. *desc-file* is preened of any existing entries referring to *target* and the new entries are appended to the file.

2.7.13.3 Using oinclude with make

make normally reads its information from description files called *makefile* or *Makefile*, or from files specified on the command line with **-f** *desc-file*. As your project grows, you may need to break your description information into a number of small files with different names. This allows you to keep track of manageable amounts of dependency information.

Trying to cite all these description files on the command line is a laborious and error-prone process. These small files can be accessed through the *include* statement, which is embedded in the main description file used by **make**. This allows the **make** statement itself to be relatively simple.

Two parallel statements in the **make** *rules* section can create the description files for headers and library routines and channel dependency output into them:

```
cpp -ttarget -Xdesc-file.hdr
ld -qtarget -Qdesc-file.lib
```

Then **make** can access the information in the description files by statements of the following form in *makefile*:

```
include  desc-file.hdr
include  desc-file.lib
```

The problem with *include* is that it does not work for the first running of **make**. The **make** program itself is going to generate the two description files. Therefore *desc-file.hdr* and *desc-file.lib* do not yet exist on the first invocation of **make**. When *include* finds missing files, it returns an error code and exits. For this reason you should use the *oinclude* statement, which will proceed as if the statement had never been called.

2.7.14 Example of a Description File

Figure 2-4 shows the description file that maintains the **make** program. The source code for **make** is spread over a number of C language source files and a Yacc grammar.

```
# Description file for the Make program

# Macro def: send to be printed
P = und -3 | opr -r2

# Macro def: source filenames used

FILES = Makefile version.c defs main.c\
        doname.c misc.c files.c\
        dosys.c gram.y lex.c gcos.c


# Macro def: object filenames used
OBJECTS = version.o main.o doname.o\
          misc.o files.o dosys.o\
          gram.o


# Macro def: lint program and flags
LINT = lint -p

# Macro def: c compiler flags
CFLAGS = -O


# make depends on the files specified
# in the OBJECTS macro definition
make:    $(OBJECTS)
# Build make with the cc program
         cc $(CFLAGS) $(OBJECTS) -o make
# Show the file sizes
         size make

# The object files depend on a file
# named defs
$(OBJECTS):  defs


# The file gram.o depends on lex.c
# uses internal rules to build gram.o
gram.o:  lex.c

# Clean up the intermediate files
clean:
         -rm *.o gram.c
         -du


# Copy the newly created program
# to /usr/bin and deletes the program
# from the current directory
install:
```

```
                @size make /usr/bin/make
                cp make /usr/bin/make ; rm make

        # Empty file "print" depends on the
        # files included in the macro FILES
        print:   $(FILES)
        # Print the recently changed files
                pr $? | $P
        # Change the date on the empty file,
        # print, to show the date of the last
        # printing
                touch print

        # Check the date of the old
        # file against the date
        # of the newly created file
        test:
                make -dp | grep -v TIME >1zap
                /usr/bin/make -dp | grep -v TIME >2zap
                diff 1zap 2zap
                rm 1zap 2zap

        # The program, lint, depends on the
        # files that are listed
        lint:    dosys.c doname.c files.c main.c misc.c
                version.c gram.c
        # Run lint on the files listed
        # LINT is an internal macro
                $(LINT) dosys.c doname.c files.c main.c \
                misc.c version.c gram.c
                rm gram.c

        # Archive the files that build make
        arch:
                ar uv /sys/source/s2/make.a $(FILES)
```

Figure 2-4. Example Description File

The **make** program usually writes out each command before issuing it.

The following output results from typing the simple command **make** in a directory containing only the source and description file:

```
  cc -O -c version.c
  cc -O -c main.c
  cc -O -c doname.c
  cc -O -c misc.c
  cc -O -c files.c
  cc -O -c dosys.c
  yacc  gram.y
  mv y.tab.c gram.c
  cc -O -c gram.c
  cc version.o main.o doname.o misc.o files.o dosys.o
     gram.o -o make
  13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars are specified in the description file, **make** uses its suffix rules to find them and issues the

needed commands. The string of digits is the result of the **size** **make** command. The @ (at sign) on the **size** command in the description file prevented writing of the command, so only the sizes are written.

The output can be sent to a different printer or to a file by changing the definition of the **P** macro on the command line as follows:

```
make print "P = print -sp"
         or
make print "P = cat >zap"
```

3.0 Chapter 3. PS/2 Assembler

Subtopics

3.1 CONTENTS

3.2 About This Chapter

This chapter discusses the assembler for the 80386 microprocessor. It discusses:

- The way statements and expressions are forme
- How the assembler labels and processes instruction
- The abbreviated instruction operands and symbols for the register
- The assembler directive
- The options used in assembling source files

3.3 Notational Conventions

The following notational conventions are used in this chapter:

... Ellipsis indicate that the preceding item may be in a list. The list is a series of the items, each separated by a comma.

. . . This notation alone on a line indicates that material not relevant to the example has been skipped.

[ ] Items enclosed in brackets are optional. Brackets enclosed in quotation marks specify actual bracket characters.

expr This specifies the appearance of an expression.

id This specifies the appearance of a symbol identifier.

The following terms which are defined in the *80386 Programmer's Reference Manual* are used in the addressing mode component, sib byte; and the field names, ss, index, and base.

3.4 Compatibility

The assembler is compatible with the UNIX System V Operating System.

The assembler supports:

Common Object File Format (COFF) module
Span-dependent instruction optimization
Arbitrarily long identifiers (80 characters are significant
Completely relocatable object module
Optional source and generated code listing
Addressing mode expressions and variables, for example

```
x=5(%ebx,%ecx,8)
```

.macro assembly, repeat block assembly, and conditional assembly directives
full floating-point suppor
very high speed assembly

There are differences between the format of assembler instruction specifications and the instructions as described in the *80386 Programmer's Reference Manual* that is published by Intel Corporation. The format published by Intel is labeled Intel format. The assembler format used here is also accepted by the UNIX assembler and is labeled UNIX format.

There are two major differences between the Intel format and the UNIX format:

The Intel format places the source operand on the right; while th UNIX format places the source operand on the left.

Most of the instructions have either byte, word, and long word forms or word and long word forms. The Intel format operand size is specified when defining storage while the UNIX format operand size is part of the instruction. Therefore the Intel format assembly language has only one move instruction: **mov**. The UNIX format assembler defines each of the three possible move instructions:

- **movb** (move byte)
- **movw** (move word)
- **movl** (move long).

This assembler operates in a 32-bit segment environment where it automatically generates a 16 bit data prefix as needed for word instructions.

3.5 80386 Architecture

As noted above, the assembler assumes that the program is intended for use in a 32-bit segment. The assembler automatically generates the data size prefix (66[16]) when a 16-bit instruction is generated. The address size prefix is never generated.

A number of instructions have a short form. Many instructions have a form with no addressing mode and an implied destination of %al, %ax, or %eax. Some instructions have a short form where the size of the immediate constant or immediate address is shorter than the size of the destination. Generally, the assembler chooses the shortest instruction. When the assembler must choose between two different abbreviations for the same instruction, the assembler selects the instructions with the shorter abbreviation (fewest components), even if the generated instruction is larger. This produces faster code when the instruction is a branch target.

The conditional branch instructions and the jump instruction have short and long forms. The assembler can process these instructions in one of three different modes. The modes are selected by specifying a command option, see Figure 3-5 in topic 3.12. The default mode tells the assembler to run one extra pass in which most of the forward references are reduced to the shortest possible form.

In all modes, it is an error when a symbol used as a branch target is redefined at a different address.

3.6 Source Statements

Subtopics
3.6.1 Statement Format
3.6.2 Character Set
3.6.3 Identifiers
3.6.4 Constants
3.6.5 Comments

3.6.1 Statement Format

An assembler statement is contained on one line of an input file. The statement may consist of a label part, an opcode or directive part, a parameter part, and a comment part. The general statement format is:

```
label: operation  parameters  /  comment
```

A statement may contain more than one label. The operation is an instruction, an assembler directive, a macro call, or an assignment statement. Multiple statements may be typed on a single line by separating each statement from the previous statement with a semicolon (;). For example:

```
.if not_terse; .string "Type exit to quit"; .endif
```

3.6.2 Character Set

Spaces and tabs separate identifiers and constants in source statements. Spaces and tabs have no other significance unless quoted. The newline character (\n), separates source lines (statements).

3.6.3 Identifiers

Identifiers (or symbols) may be composed of a combination of letters, digits, and the following characters: periods (.), dollar signs ($), and underscores (_). The first character of a symbol can not be a digit or a dollar sign ($). Upper and lower case letters are accepted and remain distinct. That is, lowercase letters (for example, a) mean something different from uppercase letters (for example, A). Assembler directives and machine instructions are defined in lowercase only. Identifiers are significant up to 80 characters.

3.6.4 Constants

Subtopics
3.6.4.1 Numeric Constants
3.6.4.2 Alphabetic Constants

3.6.4.1 Numeric Constants

A sequence of digits is a decimal numeric constant. A hexadecimal constant begins with "0" followed by a lowercase "x" (that is, 0x). The hexadecimal digits include the decimal digits and the lowercase letters "a" though "f". An octal constant begins with an "0" followed by a lowercase "o" (that is, 0o). A long floating constant begins with an "0" followed by a lowercase "f" (that is, 0f). A long floating constant may have a decimal fraction and a signed exponent.

3.6.4.2 Alphabetic Constants

There are two types of alphabetic constants: **character constants** and **string constants**.

Character Constants: A character constant is treated as an integer numeric constant with the value of the ASCII character specified. The constant is typed with single quotes (') and must specify a single character. Character constants may be used in expressions and immediate operands.

String Constants: A string constant supplies a sequence of values for the data storage directives. It consists of a sequence of character specifications enclosed in double quotes ("). String constants may contain the ASCII null character. A null character is appended to the string by the assembler. String constants may be used as parameters of storage definition instructions.

Character Translation: Character and string constants may contain all of the ASCII character set except newline characters. The ASCII backslash (\) is used within character and string constants to escape the quote marks and to specify certain control characters symbolically. A backslash followed by any other character is equivalent to the other character; in this way quotation marks or a backslash may be specified. The symbolic character specifications are:

Figure 3-1. Character Escape Sequence

ESCAPE	**CHARACTER**	**ASCII VALUE**
\0	null	0
\b	backspace	8
\t	tab	9
\n	newline	10
\r	return	13
"\""	a double quote in a string	34
'\''	a single quote in a character constant	39
'\'	a backslash	92

3.6.5 Comments

Comments begin with a slash (/) and extend to the end of the line. They can be placed at the end of a statement.

3.7 Program Segments

The output of the assembler is an object module in three segments. The three segments are called: **text, data**, and **bss**. When the link editor is used to combine several object modules, the segments from each input module are concatenated to form a single output module consisting of the combined text segments followed by the combined data segments. The operating system makes distinctions between text and data memory; the assembler, however, treats both segments identically.

Subtopics
3.7.1 Text Segment Assembly
3.7.2 Data Segment Assembly
3.7.3 Bss Segment Assembly

3.7.1 Text Segment Assembly

Assembly initially begins in the text segment. The type of the location counter is text relocatable.

3.7.2 Data Segment Assembly

The **.data** directive is available to switch to the data segment. The type of the location counter is data relocatable. The directive **.text** returns to the text segment.

3.7.3 Bss Segment Assembly

The **.bss** directive serves as a type for symbols which are assigned addresses in memory allocated when the image is loaded. This memory is initialized with zeroes. The **.comm**, **.lcomm**, or **.bss** directives allocate **bss** segment memory, and define symbols with the type **.bss** relocatable. The size of the **bss** segment is stored in the object and image files. No space (other than the space used to record the size information) is allocated for the **bss** segment in the object and image files.

3.8 Expressions

Expressions and symbols may be specified as integer expressions, or as more complex expressions which specify an entire addressing mode. The addressing mode expressions have the following general appearance:

Figure 3-2. Addressing Mode Summary

EXPRESSION	**TYPE**
reg	register mode
(reg)	register indirect
disp(reg)	register relative
addr	absolute
(base[,index][,scale])	indexed
disp[(base[,index][,scale])]	displacement indexed
[sreg:][disp][(base[,index][,scale])]	segment register override prefix displacement indexed

A reference to a label or an imported symbol (see "External Definition" in topic 3.12.11) is made with the absolute memory mode unless the instruction is a jump or call that supports a relative reference. A forward reference is assumed to be a text segment reference and may not change later to a non-memory type if it was used in a span-dependent construct.

Subtopics
3.8.1 Assignment Statements
3.8.2 Scalar Expression Operators
3.8.3 Operator Precedence
3.8.4 Expression Types
3.8.5 Type Combinations

3.8.1 Assignment Statements

An expression may be assigned to a symbol by an assignment statement in the following general form:

```
symbol=[:] expr ...
```

An expression may specify any addressing mode and that mode is generated when the symbol is used as an instruction operand. An equal sign (=) in an assignment statement defines a local constant. An equal sign followed by a colon (=:), specifies that the symbol is global. An example of these assignments is shown below.

```
a=1          / a constant
xyz=: 123    / a global constant
abc=4(%bx)   /a register relative expression
```

3.8.2 Scalar Expression Operators

A number of operators are available to form expressions. The operators are listed in the following table. The unary mode indicates that the function is recognized when the operator has only a right operand. The binary mode indicates the meaning applied when the operator has two operands.

Figure 3-3. Scalar Expression Operators

Operation	**Mode**	**Function**
$	unary	convert type to immediate
-	unary	negation
-	binary	subtraction
&	binary	logical and
^	binary	logical exclusive or
|	binary	logical or
!	binary	logical and not (0 or 1)
+	binary	addition
*	binary	multiplication
\/	binary	division
\%	binary	modulo
<	binary	less than (0 or 1)
>	binary	greater than (0 or 1)
=	binary	equality (0 or1)
<<	binary	shift left
>>	binary	shift right

3.8.3 Operator Precedence

All binary operators are implemented at the same priority. Unary operators have a higher priority than the binary operators except for the dollar sign ($) which has a lower priority than the binary operators. Expressions may be grouped with matching bracket characters.

3.8.4 Expression Types

The primary expression types are:

Constant
Constant expressions are defined ultimately from a constant or the difference between two symbols with the same type of relocation mode. The values of constant expressions are never affected by the link editor.

Immediate
Immediate expressions are produced by the dollar sign ($) operator. These expressions are referenced with the immediate addressing mode when used as instruction operands. When a label or external symbol is made immediate by the use of the dollar sign ($) operator, a relocatable expression yielding an address is produced.

Text Relocatable
These expressions are generally defined with text segment labels. These expressions are referenced with the program counter relative address mode.

Data Relocatable
These expressions are generally defined with data segment labels.

Bss Relocatable
The **.comm** and **.lcomm** directive generates **bss** relocatable symbols.

Undefined External
Symbols (that are undefined except for their appearance) in a **.globl** statement have this type. See the discussion under "Block Definition" in topic 3.12.3.

Register
The following register symbols are predefined:

8-bit	**16-bit**	**32-bit**
%al	%ax	%eax
%ah	%bx	%ebx
%bl	%cx	%ecx
%bh	%dx	%edx
%cl	%si	%esi
%ch	%di	%edi
%dl	%bp	%ebp
%dh	%sp	%esp

Complex Types

A number of complex types are composed of multiple simple types when an instruction contains several displacements or indirection steps with separate addressing constructs.

3.8.5 Type Combinations

The constant type may be combined with all operators, except where a relocatable type has been made immediate or absolute. These types and the relocatable types themselves may be combined only as follows:

+ If one operand is constant, the result is the type of the non-constant operand.

- If the second operand is constant, the result is the type of the second operand. If both operands are selected from the same type (**text**, **data**, or **bss** relocatable), the result is a constant that is the difference between the addresses.

(Relocatable types, constants and registers may be combined with the left parenthesis binary operator, "**(**", to form register indirect, register relative, or indexed expressions. A matching right parenthesis, "**)**", is required.

(reg) register indirect expression
disp(reg) register relative expression
disp(base,index,scale)
indexed expression.

The displacement specifies an absolute offset or base address in memory. The base and index fields are 32-bit registers that specify the base and index fields of the **sib** byte. The scale field is the constant 1, 2, 4, or 8 from which the **ss** field is derived. All components are optional however, either a displacement or a base register must be specified. The default scale factor is 1.

The following examples show some of the different expressions:

```
4                   / constant
4*(5-4)             / constant
$4                  / immediate
$label              / immediate relocatable
%ebx                / register
4(%ebx)             / register relative
y=8(%ebp)           / register relative
(%ebx,%ebp,8)       / indexed
```

3.9 Statement Processing

Subtopics
3.9.1 Statement Labels

3.9.1 Statement Labels

A statement may begin with one or more labels. Each label can be either a named label or a temporary label.

Subtopics
3.9.1.1 Named Labels
3.9.1.2 Temporary Labels

3.9.1.1 Named Labels

Named labels are identifiers followed by one or two colon characters. Labels defined with one colon can not be referenced outside the source module. A second colon specifies that the label is to be made external (see "External Definition" in topic 3.12.11).

Label Examples:

```
  xyz:
  abc::         .byte     1     / a global label
  l1:l2:l3:     .byte     1     / three labels on a line
```

3.9.1.2 Temporary Labels

Temporary labels consists of a non-zero numeric constant followed by a single colon character. Any number of these labels may be present even if the value of the constant is repeated. A reference to a temporary label consists of the label's constant value expressed as a decimal number, followed immediately (that is, with no space) by a lowercase f or b. The trailing letter specifies that the reference is to the nearest temporary label with the same value. The f or b further specifies a forward or backwards reference (symbols with the same value but the wrong direction are not considered nearer).

Temporary Label Examples:

```
1:     jmp     1f     / skip the next instruction
2:     jmp     1b     / select previous instruction
1:     ...     ...    / continue
```

3.10 Instruction Set

Subtopics
3.10.1 Notation and Terminology
3.10.2 Registers
3.10.3 Instructions

3.10.1 Notation and Terminology

The following abbreviations are used to describe instruction operands:

reg	A register name is required. The register name must correspond to the size of the instruction.
mem	An addressing mode for which a memory address is required.
r/m	An addressing mode for which a register or a memory address is required.
addr	A constant address is required.
immed	An immediate constant or immediate address is required.
control	A CPU control register.
debug	A CPU debug register.
test	A CPU test register.
seg	A segment/selector register.

The 80386 has no memory-to-memory addressing modes. Instructions that use addressing modes have one operand in a register and one operand in either a register or memory. These instructions are either "to memory" or "to register" instructions. Some instructions are present in both the "to memory" and "to register" form.

3.10.2 Registers

The following general register symbols are defined:

8-bit	**16-bit**	**32-bit**
%al	%ax	%eax
%ah	%bx	%ebx
%bl	%cx	%ecx
%bh	%dx	%edx
%cl	%si	%esi
%ch	%di	%edi
%dl	%bp	%ebp
%dh	%sp	%esp

The 80387 Numeric Processor register stack is specified with an expression of the form:

%st(**expr**)

where **expr** is an integer expression with a value between 0 and 7. If %st(0) is specified, it means the top of the stack; %st(1) means the next item, and so on. Some instructions, requiring two operands, that implicitly operate on the top of the stack and on a specific stack location have a reverse form. These instructions specify the implicit operand simply with %st.

The following special purpose register names are defined:

Figure 3-4. Special Purpose Registers

SYMBOL	**NAME**
%cs %ds %es %fs %gs %ss	Segment or Selector Register Names
%cr0 %cr2 %cr3	Control Register Names
%dr0 %dr1 %dr2 %dr3 %dr6	Debug Register Names

%dr7	
%tr6	
%tr7	Test Registers

3.10.3 Instructions

Numeric instructions with general operands:

```
  left vbar
  "add" rabove "adc" rabove "and" rabove "xor"
  %%% "or" rabove "sbb" rabove "sub" rabove "cmp"
  right vbar
  %
  left vbar
  "b" rabove "w" rabove "l"
  right vbar
  %
  left vbar
  <immed,r/m> labove <reg,r/m> labove <r/m,reg>
  right vbar
```

Increment and decrement instructions:

```
  left vbar
  "inc" rabove "dec"
  right vbar
  %
  left vbar
  "b" rabove "w" rabove "l"
  right vbar
  %
  left vbar
  <r/m>
  right vbar
```

The push instruction accepts an immediate, register, segment register, or memory operand. Segment operands cannot be used with the push w instruction.

```
  "push"%
  left vbar "w" rabove "l"
  right vbar
  %
  left vbar
  <immed> labove <r/m> labove <seg>
  right vbar
```

The exchange instruction swaps the constants of each operand:

```
  "xchg"%
  left vbar "b" rabove "w" rabove "l"
  right vbar
  %
  left vbar
  <reg,r/m> rabove <r/m,reg>
  right vbar
```

Move instructions defined for register and memory operands:

```
  "mov"%
  left vbar
  "b" rabove "w" rabove "l"
  right vbar
```

```
  %
  left vbar
  <immed,r/m> labove <reg,r/m> labove <r/m,reg>
  right vbar

Move instructions defined for the special CPU registers:

  "movl"%
  left vbar
  <debug,reg> labove <control,reg> labove <test,reg>
  %%%
  <reg,debug> labove <reg,control> labove <reg,test>
  right vbar

Move instructions defined for the segment registers:

  "movw"%
  left vbar
  <seg,r/m> labove <r/m,seg>
  right vbar

Move instructions defined for zero or sign extensions:

  left vbar
  "movzbw" labove "movzbl" labove "movzwl" labove "movsbw" labove "movsbl"
  labove "movswl"
  right vbar
  <r/m,reg>

String instructions:

  left vbar
  "movs" labove "cmps" labove "stos" labove "lods" labove "scas"
  %%%
  "smov" labove "scmp" labove "ssto" labove "slod" labove "ssca"
  right vbar
  %
  left vbar
  "b" labove "w" labove "l"
  right vbar


Set instructions:

  left vbar
  "seta" labove "setae" labove "setb" labove "setbe" labove "setc" labove
  "sete" labove "setg" labove "setge"
  %%%
  "setl" labove "setle" labove "setna" labove "setnae" labove "setnb"
  labove "setnbe" labove "setnc" labove "setne"
  %%%
  "setng" labove "setnge" labove "setnl" labove "setnle" labove
  "setno" labove "setnp" labove "setns" labove "setnz"
  %%%
  "seto" labove "setp" labove "setpe" labove "setpo" labove "sets" labove
  "setz"
  right vbar
  <r/m>

Rotate and shift instructions:
```

```
  left vbar
  "rcl" rabove "rcr" rabove "rol" rabove  "ror"
  ¬¬¬
  "sal" rabove "sar" rabove "shl" rabove "shr"
  right vbar
  ¬
  left vbar
  "b" rabove "w" rabove "l"
  right vbar
  ¬
  left vbar
  <"%"cl,r/m> labove <immed,r/m> labove <r/m>
  right vbar

Signed integer multiply instructions:

  "imul"%
  left vbar
  "w" labove "l"
  right vbar
  %
  Left vbar
  <immed,reg> labove <immed,r/m,reg>
  right vbar

  "imul"%
  left vbar
  %
  "b" rabove "w" labove "l"
  right vbar
  %r/m

  "imul"%
  left vbar
  "w" labove "l"
  right vbar
  %
  r/m,reg

Bitwise "and":

  "test"%
  left vbar
  "b" labove "w" labove "l"
  right vbar
  %
  left vbar
  <immed,r/m> labove <reg,r/m> labove <r/m,reg>
  right vbar

I/O instructions:

  left vbar
  "in" labove "out"
  right vbar
  %
  left vbar
  "b" rabove "w" rabove "l"
  right vbar
```

```
  %
  left vbar
  <%> labove <immed> labove <("%dx")>
  right vbar
```

Clear instructions:

```
  "clr"%
  left vbar
  "b" labove "w" labove "l"
  right vbar
  %
  <r/m>
```

Unsigned (**div**) and signed (**idiv**) division, and unsigned multiply:

```
  left vbar
  "div" labove "idiv" labove "mul"
  %%%
  "not" labove "neg"
  right vbar
  %
  left vbar
  <reg,r/m> labove <r/m,reg>
  right vbar
```

Stack pop instructions operate on a register, memory or segment register operand :

```
  "pop"%
  left vbar
  "w" rabove "l"
  right vbar
  %
  left vbar
  <seg> labove <r/m>
  right vbar
```

A number of instructions require an operand to be in memory and do not accept a register as operand 1:

```
  left vbar
  "bound"%% "lea" rabove "lds" rabove "les"
  %%%
  "lfs" labove "lgs" labove "lss"
  right vbar
  %
  left vbar
  "w" rabove "l"
  right vbar
  %mem,r/m
```

Note: The **bound** instruction generates a SIGPRE exception if the bound test fails.

Bit scan instructions:

```
  left vbar
  "bsf" rabove "bsr"
```

```
  right vbar
  %
  left vbar
  "w" labove "l"
  right vbar
  %
  r/m, reg

Bit test instructions:

  left vbar
  "btc" labove "bt"
  %%%
  "btr" labove "bts"
  right vbar
  %
  left vbar
  "w" labove "l"
  right vbar
  %
  left vbar
  <immed,r/m> labove <reg,r/m>
  right vbar

Double-precision shift instructions:

  left vbar
  "shld" labove "shrd"
  right vbar
  %
  left vbar
  "w"labove "l"
  right vbar
  %
  left vbar
  <immed,reg,r/m> labove <reg,r/m>
  right vbar


Instruction prefix bytes defined as one-byte instructions:

  left vbar
  "addr16" labove "data16" labove "lock" labove "rep"
  %%%
  "repe" labove "repne" labove "repz" labove "repnz"
  %%%
  "seg_cs" labove "seg_ds" labove "seg_es" labove "seg_fs"
  %%%
  "seg_gs" labove "seg_ss"
  right vbar

Instructions with no operands:

  left vbar
  "aaa" labove "aad" labove "aam" labove "aas" labove "cbtw"
  labove "clc"
  %%%
  "cld" labove "cli cltd" labove "clts" labove "cmc"
  labove "cwtd" labove "cwtl"
  %%%
```

```
  "daa" labove "das" labove "fwait" labove "hlt"
  labove "into" labove "iret"
  %%%
  "lahf" labove "leave" labove "nop"
  labove "popa" labove "popal" labove"popaw"
  %%%
  "popf" labove "popfl" labove "popfw"
  labove "pusha" labove "pushal" labove "pushaw"
  %%%
  "pushf" labove "pushfl" labove "pushfw"
  labove "sahf" labove "stc" labove "std"
  %%%
  "sti" labove "wait" labove "xlat"
  right vbar

Conditional jump instructions:

  left vbar
  "ja" labove "jae" labove "jb" labove "jbe"
  %%%
  "jc" labove "je" labove "jg" labove "jge"
  %%%
  "jl" labove "jle" labove "jna" labove "jnae"
  %%%
  "jnb" labove "jnbe" labove "jnc" labove "jne"
  %%%
  "jng" labove "jnge" labove "jnl" labove "jnle"
  %%%
  "jno" labove "jnp" labove "jns" labove "jnz"
  %%%
  "jo" labove "jp" labove "jpe jz" labove "jpo"
  %%%
  "js"
  right vbar
  %
  addr

Jump and Call instructions:

  left vbar
  "call" labove "jmp"
  right vbar
  %
  left vbar
  <adjust(d 3) '*'%%r/m> labove <addr>
  right vbar

Long jump and long call instructions. In the *r/m form, the operand is a
far pointer. In the immediate form, the first operand is the selector for
a descriptor and the second operand is the offset into the new segment.

  left vbar
  "lcall" labove "ljmp"
  right vbar
  %
  left vbar
  <adjust(d 3) '*'%%r/m> labove <immed,immed>
  right vbar
```

3.11 Instructions for the 80387 Numeric Processor

```
Multiply and add instructions with register stack operands:

  left vbar
  "fadd" labove "fmul"
  right vbar
  ¬
  left vbar
  <"%st,%st"(i)> labove <"%st,"(i),"%st">
  right vbar

Multiply and add instructions with a memory operand and the pop option:

  left vbar
  "faddp" labove "fmulp"
  right vbar
  ¬
  "%st,%st"<(i)>

Multiply and add instructions with a memory operand:

  "f"%
  left vbar
  "add" labove "mul"
  right vbar
  %
  left vbar
  "l" labove "s"
  right vbar
  %memop

  "fi"%
  left vbar
  "add" labove "mul"
  right vbar
  %
  left lbracket
  "l"
  right rbracket
  %memop

Divide and subtract instructions with a single memory operand:

  left vbar
  "fidiv" labove "fisub"
  right vbar
  %
  left lbracket
  "r"
  right rbracket
  %
  left lbracket
  "l"
  right rbracket
  %memop

  left vbar
  "fdiv" labove "fsub"
  right vbar
```

```
  %
  left lbracket
  "r"
  right rbracket
  %
  left vbar
  "l" labove "s"
  right vbar
  %memop

Divide and subtract instructions with register stack operands:

  left vbar
  "fdiv" labove "fsub"
  right vbar
  ¬
  left lbracket
  "r"
  right rbracket
  ¬
  left vbar
  <"%st,%st"(i)> labove <"%st"(i),"%st">
  right vbar

  left vbar"
  "fdiv" labove "fsub"
  right vbar
  ¬
  left lbracket
  "r"
  right rbracket
  "p"¬¬
  "%st,%st"(i)

Comparison instructions with a single register operand:

  "f"¬
  left lbracket
  "u"
  right rbracket
  ¬"com"¬
  left lbracket
  "p"
  right rbracket
  ¬"%st"(i)


Numeric processor instructions with one register operand:

  "ffree"¬
  "%st"(i)

Numeric processor instructions with no operands:

  left vbar
  "f2xm1" labove "fabs" labove "fchs" labove "fclex" labove "fcompp"
  %%
  "fcos" labove "fdecstp" labove "fincstp" labove "finit" labove "fld1"
  %%
  "fldl2e" labove "fldl2t" labove "fldlg2" labove "fldln2" labove "fldpi"
```

```
  %%
  "fldz" labove "fnclex" labove  "fninit" labove "fnop" labove "fpatan"
  %%
  "fprem" labove "fprem1" labove "fptan" labove "frndint" labove "fscale"
  %%
  "fsetpm" labove "fsin" labove "fsincos" labove "fsqrt" labove "ftst"
  %%
  "fucompp" labove "fwait" labove "fxam" labove "fxtract" labove
  "fyl2x"
  %%
  "fyl2xp1"
  right vbar
```

Numeric processor instructions with one memory operand:

```
  left vbar
  "fld" labove "fstp"
  right vbar
  %
  left vbar
  "s" labove "l" labove "t"
  right vbar
  %memop
```

Numeric processor instructions with short real and long real formats:

```
  left vbar
  "fst" labove "fcom" labove "fcomp"
  right vbar
  %
  left vbar
  "l" labove "s"
  right vbar
  %memop
```

Numeric processor instructions with word, long word, and very long word formats:

```
  left vbar
  "fild" labove "fistp"
  right vbar
  %
  left vbar
  "-" labove "l" labove "ll"
  right vbar
  %memop
```

Numeric processor Binary Coded Decimal load and store:

```
  left vbar
  "fbld" labove "fbstp"
  right vbar
  %memop
```

Numeric processor word and long word integer instructions with one memory operand:

```
  left vbar
  "fist" labove "ficom" labove "ficomp"
  right vbar
```

```
  %
  left lbracket
  "l"
  right rbracket
  %memop
```

Other numeric processor instructions with one memory operand:

```
  left vbar
  "fsave" labove "fnsave" labove "fstcw" labove "fnstcw"
  %%
  "fstenv" labove "fnstenv" labove "fstsw" labove "fnstsw"
  %%
  "fldcw" labove "fldenv" labove "frstor"
  right vbar
  %memop
```

3.12 Assembler Directives

Assembler directives are specified in a manner similar to the specification of instructions. Directives control options of the assembler or format and generate data for the code segments. Certain directives establish or alter the definitions of symbols.

The directives themselves are symbols with the type of directive and a predefined value which specifies the particular directive. Both directives and instructions appear between labels and operands. The assembler assigns a specific type to such predefined symbols and searches only for this type between labels and operands. Therefore, labels can have the same names as instructions and directives.

Figure 3-5. Directive Summary

Directive	Function	Additional Information
.align	Adjust location counter to boundary.	"Alignment Definition" in topic 3.12.1
.bcd	Defines a binary coded decimal value.	"Storage Definition" in topic 3.12.6
.blkb	Reserves a block of bytes.	"Block Definition" in topic 3.12.3
.blkd	Reserves a block of double words.	"Block Definition" in topic 3.12.3
.blkf	Reserves a block of real numbers.	"Block Definition" in topic 3.12.3
.blkl	Reserves a block of longs.	"Block Definition" in topic 3.12.3
.blkw	Reserves a block of words.	"Block Definition" in topic 3.12.3
.bss	Switches to the bss section.	"Program Section (Segment) Control" in topic 3.12.2 and "Block Definition" in topic 3.12.3
.byte	Defines bytes.	"Storage Definition" in topic 3.12.6
.comm	Defines a common block	"Block Definition" in topic 3.12.3
.data	Switches to the data section.	"Program Section (Segment) Control" in topic 3.12.2
.double	Defines double reals.	"Storage Definition" in topic 3.12.6
.dsect	Defines a record.	"Record Definition" in topic 3.12.5

.else	Reverses condition to **.if**.	"Conditional Block" in topic 3.12.10
.elseif	Reverse condition to **.if** plus a new condition.	"Conditional Block" in topic 3.12.10
.end	Ends a record	"Record Definition" in topic 3.12.5
.endif	Ends **.if** block.	"Conditional Block" in topic 3.12.10
.endm	Ends a macro.	"Macro Definition" in topic 3.12.8
.endr	Ends a repeat block.	"Repeat Block" in topic 3.12.9
.enum	Defines a series of identifiers.	"Enumeration" in topic 3.12.7
.even	Adjusts the location counter to an even boundary.	"Alignment Definition" in topic 3.12.1
.extern	Imports or exports a symbol.	"External Definition" in topic 3.12.11
.float	Defines real numbers.	"Storage Definition" in topic 3.12.6
.globl	Alternate name for **.extern**.	"External Definition" in topic 3.12.11
.ident	Defines a string in the comment section.	"Comment Section" in topic 3.12.4
.if	Enters a conditional block.	"Conditional Block" in topic 3.12.10
.lcomm	Defines a local zero-filled object.	"Block Definition" in topic 3.12.3
.list	Enable the source listing.	"External Definition" in topic 3.12.11
.long	Defines long numbers.	"Storage Definition" in topic 3.12.6
.macro	Defines a macro.	"Macro Definition" in topic 3.12.8
.nlist	Disables the source listing.	"External Definition" in topic 3.12.11
.noopt	Turns off the span optimization.	"Optimization" in topic 3.12.13
.optim	Turns on the span optimization.	"Optimization" in topic 3.12.13

.rept	Repeats the following statements.	"Repeat Block" in topic 3.12.9
.set	Assigns a value.	"Assignment" in topic 3.12.12
.string	Defines a string.	"Storage Definition" in topic 3.12.6
.text	Switches to the text section.	"Program Section (Segment) Control" in topic 3.12.2
.value	Defines words.	"Storage Definition" in topic 3.12.6
.version	Defines version strings.	"Comment Section" in topic 3.12.4

Subtopics
3.12.1 Alignment Definition
3.12.2 Program Section (Segment) Control
3.12.3 Block Definition
3.12.4 Comment Section
3.12.5 Record Definition
3.12.6 Storage Definition
3.12.7 Enumeration
3.12.8 Macro Definition
3.12.9 Repeat Block
3.12.10 Conditional Block
3.12.11 External Definition
3.12.12 Assignment
3.12.13 Optimization

3.12.1 Alignment Definition

.align .byte **.align** .value **.align** .long **.align** .float **.align** .double **.even** **.align expr** [,**expr**]	These directives advance the location counter to an addressing boundary specified by the keyword or constant expression. The boundary is such that the location counter is evenly divisible by the first expression or the size of the object referred to by the keyword.

The location counter is advanced to the boundary specified by the first parameter. As the location counter is advanced, the segment is normally filled with **nop** instructions. A second **expr** parameter is optionally accepted and is then used as the fill value.

All expressions must be constant and must be defined in pass one.

The **.even** directive is equivalent to a **.align** 2 directive.

3.12.2 Program Section (Segment) Control

.text **.data** **.bss**	These directives divert generated object code into the specified program section.

The assembler supports two program sections for initialized object code. These sections are named **text** and **data**. The **bss** section, the third program section, contains only space that has not been initialized. Each program section is assembled separately and the final object contains all the text section object code followed by all of the data section code.

.text Assembly continues in the text segment.
.data Assembly continues in the data segment.

Note: The **-R** option suppresses the **.data** directive and assembles all statements in the text segment. (See "Command Format" in topic 3.13 for additional information).

3.12.3 Block Definition

.blkb expr **.blkw expr** **.blkl expr** **.blkf expr** **.blkd expr**	These directives reserve space for **expr** objects of the indicated size. If no expression is present, one such location is reserved. In the **text** or **data** sections, bytes of zeros are generated.

The block directives reserve space for objects of the indicated size. When a block directive is used within a **.dsect** (record definition), the location counter is advanced by the required amount of space. In the **text** and **data** sections, bytes of zeros are assembled in the object module.

When independent and uninitialized or zero-initialized variables are specified with the block directives (or with directives such as **.byte** 0) the object file must contain the actual bytes of clear data. The following directives are more appropriate for the definition of zero-filled object.

.comm id,expr **.lcomm id,expr** **.bss id,expr**	These directives reserve space in the memory image without requiring space in the object file.

The type of the symbol is set to an undefined external type and the value is set to **expr**. Such symbols may be redefined as program section symbols in the module where they appear or in other modules that will be combined with the link editor. If no such program section symbols appear, the linker defines undefined symbols with non-zero values as common blocks in the final image file's **bss** segment. The length of the common block is taken from the symbol's values (which would normally be the symbol's address). When many identically named common blocks are present, the linker defines a single block with the size of the highest valued common symbol.

This processing makes it possible to declare a common block in many different modules, and to initialize that block (optionally) in only one module. When a module initializes a symbol that is a common block in another module, the symbol is defined in the **text** or **data** segment, and the common references select the **text** or **data** object after link editing.

The cost in space and time of using a common block is similar to a reference to any other type of symbol. It is reasonable to declare every external variable (even integers and characters) as a separate common block.

The symbol is defined as a local **bss** location and **expr** bytes are reserved. This directive is useful for allocating objects which are not initialized (or initialized to zero) and not exported.

3.12.4 Comment Section

.version string	These directives put the string values in the comment section.
.ident string	

The assembler supports an optional comment section.

3.12.5 Record Definition

.dsect (labels) (directives) (labels) (assignments) . . . **.end**	This directive is used to define records and structures. The location counter begins at zero in each **dsect**, and is incremented according to block directives.

The purpose of the **dsect** directive is to assign increasing constant values to the labels. The **dsect** is a dummy program section where code may not be generated. The only tasks allowed are: assigning labels, aligning directives and blocking directives. A dsect begins with the **.dsect** directive and ends with the **.end** directive. It is useful for defining records, stacks, and frame structures.

3.12.6 Storage Definition

.byte **.string** **.value** **.long** **.float** **.double**	These directives initialize bytes of the **text** or **data** segments. The directives accept a list of expressions or string constants. Each expression may be preceded by a repeat count.

The general format of a storage definition directive is:

directive [[(expr)] expr] ...

The repeat count is an optional expression enclosed in parenthesis (). A list of expressions follows. Each expression allocates a location of the specified size. A string constant may appear in an expression. In this case, each character specification within the string constant generates an object of the size specified by the directive.

When a repeat count is present, it must be constant and defined when in pass one. The next expression is assembled the indicated number of times.

.byte	Assemble bytes
.string	Equivalent to **.byte** plus a final zero byte
.value	Assemble 16-bit short words
.long	Assemble 32-bit long words
.float	assemble 32-bit floating constants
.double	assemble 64-bit floating constants

.bcd expr	This directive generates a 10-byte (80-bit) decimal string in the current section. This directive is not valid for the **.bss** section.

Examples using the storage directives follow.

```
  .byte     '*','*','*'      / store three stars
  .byte     (3)'*'           / store three stars
  .byte     "***"            / store three stars
  .byte     (3)"***"         / store three stars
  .double   1                / store double 1
  .value     1,2,3           / store three words
  .value     (3)1,(3)2,(3)3  / store nine words
  .float    Of3.14159        / store a 4 byte real
  .double   Of3.14159265     / store an 8 byte real
  .string   "hello"          / store a 6 byte string
```

3.12.7 Enumeration

.enum id...	This directive is used to define a list of symbols with scalar values beginning at zero and increasing by one over the list.

The general format of the **.enum** directive is:

.enum[symbol [=expr]] ...

The directive defines symbols as constants with increasing values. Normally, the first symbol's value is zero with subsequent values each greater by one. Any symbol may be followed by an assignment to reset the sequence to an arbitrary expression.

Examples using the **.enum** directives follow.

```
  .enum   a,b,c     ; define a=0,b=1,c=2
  .enum   x=5,y,z    ; define x=5,y=6,z=7
```

3.12.8 Macro Definition

.macro name [list] . . . **.endm**	This directive enters a macro definition. Assembler statements are collected until a matching **.endm** directive is processed.

The **.macro** directive assigns a name and a local parameter list to a sequence of assembler statements. The parameter list consists of identifiers separated by commas or white space.

After a matching **.endm** directive is processed, the assembler recognizes the name of the macro and substitutes the saved assembler statements. This procedure is said to invoke the macro and is known as **.macro** expansion. Actual parameters are supplied when the macro is invoked, and there must be the same number of actual parameters as there are identifiers in the parameter list of the macro definition.

Actual parameters are separated by commas.

The **.endm** directive must be the first symbol on its line; no labels are permitted.

During macro expansion, all references to a parameter of the definition are replaced by the corresponding actual parameter. The resulting assembler statement is not scanned for further parameter matches. If one macro calls another, the parameters of the first invocation are hidden from that of the inner.

A macro may itself contain macro definitions. In this case the inner definition is processed only when the macro is later expanded.

When a macro is invoked, the name of the macro appears in the listing. The expansion of the macro and the correspondingly generated object code are then listed. The directives, **.nlist** and **.macro**, can be used to disable the listing of the macro expansion. (See "External Definition" in topic 3.12.11).

Examples using the **.macro** directives follow.

```
  .macro checkvalue value errormsg
                  cmpl    $0, value      /if value is zero,
                  jnz     $cv4
                  pushl   $msg
                  call    error         /print error message
                  jmp     cv4           /and continue
  msg:    string errormsg
  cv4:
  endm
          . . .
          checkvalue %-0X8(%ebp), "second value must not be zero-n"
          . . .
```

3.12.9 Repeat Block

.rept expr . . . **.endr**	This directive specifies a block of assembly statements which are to be repeated the number of times specified by **expr**.

The expression must be constant and must be defined during pass one. Repeat blocks may occur within repeat blocks. In this case, the inner repeat block is expanded once for each expansion of the next outer block. The repeat count of an inner block is evaluated at each expansion of the inner block.

The **.endr** directive must be the first symbol on its line; no labels are permitted.

Repeat blocks may be contained within macro definitions, or definitions may be contained within blocks, but no other overlap is possible.

Object code resulting from the assembly of a repeat block never appears in the listing. The source text of a repeat block appears in the listing no more than once.

Examples using the **.rept** directives follow.

```
  .macro factorial num
                  subl  %esi, %esi              /fact = 0
                  movl  $num, %ebx              /tmp = num
  .rept num
                  leal  (%ebx, %esi), %esi      /fact = fact + tmp
                  leal  -0x1 (%ebx), %eax
                  movl  %eax, %ebx              /tmp = tmp - 1
  .endr
  .endm
                  ...
                  factorial 20
                  ...
```

3.12.10 Conditional Block

.if expr **.else** **.endif** **.if expr** **.elseif expr** **.endif**	The **.if** directive specifies a block of assembly statements which are to be assembled only if **expr** is non-zero. The reverse condition to the **.else** block, and the reverse of the condition plus a new condition applies to an **.elseif** block.

The **expr** is evaluated. It must be constant and defined within pass one. If its value is non-zero, the block of statements is assembled normally. Otherwise, the generation of code, the definition of symbols and labels, and the processing of directives are suppressed until a matching **.endif** is processed. The **.else** directive may be used to reverse the condition and begin assembling statements only if the matching **.if** was false. The **.elseif** directive is equivalent to a **.else** followed by a second **.if**, except that only one **.endif** is required to terminate the block.

Conditional blocks may occur within conditional blocks.

The conditional block is always listed, but no object code listing appears for blocks which are not assembled.

3.12.11 External Definition

.extern id ... **.globl id...**	These directives are used to import symbols defined in other modules and to export symbols defined within the assembly module.

The **.extern** directive defines a list of symbols as external. If such a symbol is defined within the module as a constant or program section symbol, the effect is to make the value and type available to the link editor. Otherwise, the symbol is an undefined-external, and the linker is instructed to import the symbol and relocate any references to it.

The list of symbols may be separated with commas, spaces, or tabs.

The **.extern** and **.globl** directives are equivalent.

Examples using the **.extern** directives follow.

```
  .extern start
  .extern sin,cos,fun
  .extern proc1 proc2
```

3.12.12 Assignment

.set id,expr	This directive sets the value of the symbol **id** to **expr**. This is equivalent to an assignment.

Use the **.set** directive as shown below:

```
.set mysymb, 4
```

This will set the value of the symbol, **mysymb**, to 4 in that file's symbol table. It is equivalent to specifying the assignment:

```
mysymb=4
```

3.12.13 Optimization

.noopt	These directives are used to turn off and on span optimizations.
.optim	

Note: The assembler passes these directives in pass1 and uses the last one specified for assembling the entire module. Span optimizations cannot be selectively turned on and off for sections of a module.

3.13 Command Format

Syntax

```
as[[ option...] [input_file...]]...
```

The assembler combines each specified input file and produces a single output object module. The name of the output file is based on the last input file name. If the input file name ends in a ".s," the output file name is created by replacing the ".s" with a ".o" extension. Otherwise the ".o" extension is appended to the input file name. If the input file name is greater than 12 characters, the file name is truncated to 12 characters and the ".o" extension is appended.

A number of options may be specified.

-a Does not automatically import any symbols that are referenced in but are otherwise undefined. Issues an error message for this case.

-R Suppresses any **.data** directives; all code is assembled in the text segment.

-l [file] Generates a source listing. If the optional file is specified, the source listing is written to that file. Do not leave a space between the **-l** and the file name.

-o file Sets the name of the output module to **file**.

-s0 Generates the long form for all forward references and the short form, where possible, for backward references.

-s1 Runs one extra pass, in which most of the forward references are reduced to the shortest possible form.

-s2 Runs as many passes as are necessary to generate the short form for all qualifying forward references.

4.0 Chapter 4. AIX/370 Assembler

Subtopics

4.1 Contents

4.2 About This Chapter

This chapter explains the differences that exist between the Basic Assembler Language for System/370 and the Basic Assembler Language for AIX/370. Refer to the *OS/VS-DOS/VS-VM/370 Assembler Language* for additional information.

4.3 Using the Assembler Language

The **as** (assembler) has the form:

```
as [-options] filename
```

The following table shows the options that are valid for the AIX/370 Assembler Language:

Option	Description	Default
-o objfile	The object of the assembly is left in the file *objfile*.	*filename*.**o**
-1 listfile	Creates a file listing.	none
-T dnu	Sets the origin of the text segment to the decimal number dnu.	10 000
-D	Uses different version of the line and bcall built-in macros.	none
-i int **n**	Prints every **n**th line.	none
-n n	Size of the symbol table in "buckets."	none
-t	Times the assembler. Also counts and prints the number of lines processed and the number of lines expanded from macros.	none
-V	Causes the version number of the assembler being run to be written on standard error.	none
-Xa	Uses the xa instruction set.	none
-b	Specifies base register.	none
-C	Fixed format inspection (for example, card format).	none
-s n	Tab size. Used only with the -C option.	none
-dl	Removes line number entries from the symbol table.	none.
-m	Recognized but not supported.	none

In addition, you can use **cc** command (compiler) to compile assembler language files.

Note: The AIX Assemblers are intended simply to support the compilers. They do not support all the functions of assemblers specifically designed for assembly language programming. **dbx** can disassemble compiler generated code, but it uses the AIX assembler to do this. Therefore, some restrictions apply to assembly code generated by

this method.

4.4 Differences

Subtopics
4.4.1 Lowercase Support
4.4.2 Hex Numbers
4.4.3 Segments
4.4.4 Escapes
4.4.5 Preprocessor Support
4.4.6 Identifier Symbols
4.4.7 Macro Definitions
4.4.8 CSECT Symbols
4.4.9 Free--form Input

4.4.1 Lowercase Support

The lowercase support allows any uppercase program to be mapped in its entirety to lowercase with minimal effect on the resulting program. The differences that are detectable in the **a.out** file are:

Literals where lowercase letters are represented as C' . . .

Symbols in the symbol table are in the same case as they were in th source program.

Subtopics
4.4.1.1 Implications

4.4.1.1 Implications

Operation codes: All operation codes are mapped arbitrarily to uppercase before recognition search begins. For example, you cannot define two macros with the same name (one in lowercase and one in uppercase; such as BILL and bill).

Symbols: Symbols are entered into the symbol table as they are found in the source program. Therefore, you can have two distinct symbols with BILL and bill.

Attributes: Attributes may appear in lowercase. The attribute t' returns the lowercase equivalent of the attribute T'. For example,

```
AIF    (T'&X EQ 'U'). . .
```

works as well as

```
aif    (t'&x eq 'u'). . .
```

System variable symbols: When a variable symbol such as **&SYSLIST** is supported, then **&syslist** is also supported. Only one version of the variable actually exists (uppercase); however, the lookup procedure searches for variable symbols beginning with **sys** twice (raw form and once after mapping to uppercase).

Miscellany: Lowercase may also be used in self-defining terms, DC constants (such as c'#'), and in hex constants (such as X'a0') with the same meaning as its uppercase equivalent.

4.4.2 Hex Numbers

The '0x' convention is not honored because its effect can be obtained with the standard x'...' convention.

4.4.3 Segments

An AIX/370 **a.out** file is partitioned into three segments: **text**, **data**, and **bss**.

The segments and their associated pseudo-ops are as follows:

Segment 'Labeled pseudo-ops'

text CSECT

data PSECT

bss ZSECT

These pseudo-ops can be used to provide the effect of multiple location counters (as they can be in the standard Basic Assembler Language). Thus:

```
A    CSECT
    .
    . code A1
    .
B    PSECT
    .
    . code B1
    .
C    CSECT
    .
    . code C1
    .
A    CSECT
    .
    . code A2
    .
    END
```

places A2 after A1 and before C1 in the text region and places B1 in the data region.

The Dsect pseudo-op is also implemented. The same label may appear on only one SECT-type pseudo-op (CSECT, PSECT, ZSECT, or DSECT). By default the **null** label is associated with the CSECT at the start of the program. Therefore, a null-labeled PSECT, ZSECT, OR DSECT is in error.

4.4.4 Escapes

Within character literals (for example, tokens of the form C'...') the backslash (\) character is treated as an escape and the usual C escapes are honored (such as, \n, \b, \t, etc.). Note that the only two escape mechanisms are the double quote ('') for compatibility with BAL, and the C escapes.

4.4.5 Preprocessor Support

The output of the preprocessor normally contains #'s identifying original source line numbers. These are understood by the assembler. However, a symbol may not consist of the # character alone.

4.4.6 Identifier Symbols

In addition to lower and uppercase letters, numerics, and national characters ($, @, #), identifiers may contain the underscore (_) character.

4.4.7 Macro Definitions

Macros need not appear before the first CSECT (as is required in BAL) but only before their first use. Their names may conflict with and override existing instructions and pseudo-ops.

4.4.8 CSECT Symbols

CSECTs do not automatically become externally known, so the function of multiple location counter usage and external identification can be separated. This can always be done with an ENTRY statement.

4.4.9 Free--form Input

Except for the **-C** and **-s** options, a complete break is made from the card image processing. Therefore, a continuation is not indicated by a non-blank in column 72, and columns 73 - 80 are not used for sequence identification. In addition, blank lines are permitted and a new line terminates a statement.

4.5 Restrictions

Subtopics
4.5.1 Op-codes
4.5.2 Pseudo-ops
4.5.3 Attributes
4.5.4 Assembler Symbols
4.5.5 Macros
4.5.6 Conditional Assembly Instructions

4.5.1 Op-codes

The AIX/370 Assembler Language does not recognize the following op-codes:

CFC
LAB
MVCIN
SIE
STAM
UPT

4.5.2 Pseudo-ops

The AIX/370 Assembler Language does not recognize the following pseudo-ops:

AMODE
ACTR
AREAD
CCW0
CCW1
COM
COPY
CXD
DXD
ICTL
ISEQ
LOCTR
MHELP
OPSYN
POP
PUNCH
PUSH
REPRO
RMODE
START

The following pseudo-ops are recognized but no action is taken:

EJECT
PRINT
SPACE
TITLE

The following pseudo-ops have restrictions:

DC Modifiers **S** and **E** are not supported; nor are types **L**, **P**, **Z**, **S**, or **Q**

EQU The optional length and type arguments are ignored.

MEND Cannot contain a sequence symbol.

MNOTE The severity number, if present, is ignored.

WXTRN Is treated as EXTRN.

4.5.3 Attributes

The following attributes are not supported:

S'
I'

The following attributes are partially supported:

K'
N'
T'

Note: These three attributes can only be applied to variable symbols (&-symbols).

The following two attributes can only be applied to symbols previously encountered when used in a conditional assembly statement:

T'
L'

4.5.4 Assembler Symbols

Set Symbols: Arrays of set symbols are not allowed.

System Variable Symbols: The following system variable symbols are supported:

&SYSECT
&SYSLIST
&SYSNDX

4.5.5 Macros

An operation field within a macro may not contain a variable symbol.

4.5.6 Conditional Assembly Instructions

Branching backwards via conditional assembly instructions (AGO and AIF) is not permitted in open code.

5.0 Chapter 5. Linking Your Programs - Using the ld Command

Subtopics

5.1 Contents

5.2 About This Chapter

This chapter describes how the link editor creates executable object files and processes symbolic debugging information. It also contains information on how to use the link editor and the link editor command language.

5.3 Link Editor

The link editor creates executable object files by combining object files, performing relocation, and resolving external references.

It also processes symbolic debugging information. The inputs to the link editor are relocatable object files produced by any of the following:

The C compiler **cc**
The assembler **as**
The previous link editor run

The link editor combines these object files to form either a relocatable or an absolute (executable) object file.

Although you can control the link-edit process as described in "Link Editor Command Language" in topic 5.10 you may not require this degree of flexibility. The **ld** command language and the description in *AIX/370 Command Reference* manual are sufficient to understand how to use the command.

The **ld** command language allows you to:

Specify the memory configuration of the machine

Combine object file sections in particular fashions

Specify the files to be bound to specific addresses or within specifi portions of memory.

Define or redefine global symbols at link edit time

5.4 Memory Configuration

Virtual memory is partitioned into configured and non-configured memory. The default is to treat all memory as configured. Non-configured memory is treated as "reserved" or "unusable" by the link editor. Nothing can ever be linked into non-configured memory. Therefore, specifying a certain memory range to be non-configured is one way of marking the addresses (in that range) "illegal" or "nonexistent" with respect to the linking process. You must explicitly specify memory configurations other than the default.

Note: Unless otherwise specified, all information is this chapter is described with respect to the configured address space.

5.5 Section

A section of an object file is the smallest unit of relocation and must be a contiguous block of memory. A section is identified by starting address and a size. Information describing all the sections in a file is stored in "section headers" at the start of the file. Sections from input files are combined to form output sections that contain executable text, data, or a mixture of both. Although there may be holes or gaps between input sections and between output sections, storage is allocated contiguously within each output section and may not overlap a hole in memory.

5.6 Addresses

The physical address of a section or symbol is the relative offset from address zero of the address space. The physical address of an object is not necessarily the location at which it is placed when the process is executed; because of paging, the address is with respect to address zero of the virtual space, and the system performs another address translation.

5.7 Binding

It is often necessary to have a section begin at a specific, predefined address in the address space. The process of specifying this starting address is called ***binding***, and the section in question is said to be "bound to" or "bound at" the required address. While binding is most relevant to output sections, it is also possible to bind global symbols with an assignment statement in the **ld** command language.

5.8 Object File

Object files are produced both by the assembler (typically as a result of calling the compiler) and by the link editor. The link editor accepts relocatable object files as input and produces an output object file that may or may not be relocatable. Under special circumstances, the input object files given to the link editor can also be absolute files.

Files produced from the compiler/assembler always contain the following three sections:

.text	Contains the instruction text (for example, executable instructions).
.data	Contains initialized data variables.
.bss	Contains non-initialized data variables.

Global non-initialized data is treated as common and the section (**.bss** or **.data**) is not bound until link-time.

5.9 Using the Link Editor

To invoke the link editor, issue the command:

```
ld [options] filename1 filename2...
```

Files passed to the link editor must be one of the following:

Object file

Archive libraries containing object file

Text source files containing **ld** directives.

The link editor uses the "magic number" (in the first two bytes of the file) to determine which type of file is encountered. If the link editor does not recognize the magic number, it assumes the file is a text file containing link editor directives and attempts to parse it.

Input object files and archive libraries of object files are linked together to form an output object file. If there are no unresolved references, this file is executable.

Note: In this chapter, an input file containing directives is referred to as an **ifile**, while object files have the following form:

```
name.o
```

The names of actual input object files need not follow this convention.

If you want to link the object files **file1.o** and **file2.o**, enter the following command:

```
ld file1.o file2.o
```

No directives to the link editor are needed. If no errors are encountered during the link edit, the output is placed in the default file **a.out**. The sections of the input files are combined in order. That is, if **file1.o** and **file2.o** each contains the standard sections **.text**, **.data**, and **.bss**, the output object file will contain them as well. The output **.text** section is a concatenation of **.text** from **file1.o** and **.text** from **file2.o**. The **.data** and **.bss** sections are formed similarly. On AIX/370, the output **.text** section is then bound at address 0x0000a8 except with option **-ss**. The output **.data** and **.bss** sections are link-edited together into contiguous addresses (the particular address depending on the particular processor).

Instead of entering the names of files to be link edited (as well as **ld** options on the command line), this information can be placed into an **ifile**. The **ifile** can be passed to **ld**. For example, if you frequently link the object files file **1.o**, **file 2.o**, and **file3.o** with the same options **f1** and **f2** by entering the command:

```
ld -m -r file 1.o file2.o file3.o
```

you can alternatively create an **ifile** containing the statements:

```
-m
-r
file1.o
```

```
file2.o
file3.o
```

Then you only need to issue the command:

```
ld ifile
```

You can specify some of the object files to be link-edited in the **ifile** and others on the command line and/or some options in the **ifile** and others on the command line. Input object files are link edited in the order they are encountered, whether this occurs on the command line or in an **ifile**. As an example, if a command line were:

```
ld file1.o ifile file2.o
```

and the **ifile** contained:

```
file3.o
file4.o
```

then the order of link editing would be: **file1.o**, **file3.o**, **file4.o**, and **file2.o**. Note from this example that an **ifile** is read and processed immediately upon being encountered in the command line.

Options may be interspersed with file names both on the command line and in an **ifile**. The ordering of options is only significant for the **-l** and **-L** options when specifying libraries. The **-l** option is a shorthand notation for specifying an archive library (an archive library is a collection of object files). As with object files, libraries are searched as they are encountered. The **-L** option specifies an alternative directory for searching for libraries. Therefore, to be effective, a **-L** option must appear before any **-l** options.

All **ld** options must be preceded by a hyphen (-) whether in the **ifile** or on the **ld** command line. Options that have an argument (except for the **-l** option) are separated from the argument by white space (blanks or tabs). The following **ld** flags may be specific on AIX/370.

-e *sym* Defines the primary entry point of the output file to be the symbol given by the argument *sym*. See "Changing the Entry Point" in topic 5.11.1 for a discussion on how the option is used.

-f *fill* Sets the default fill value. This value is used to fill holes formed within output sections. Also, it is used to initialize input **.bss** sections when they are combined with other non-**.bss** input sections. The argument *fill* is a 2-byte constant. If this option is not used, the **default** fill-value is zero.

-l *[key]* Specifies an archive library file as input. The optional argument is a character string (less than 7 characters) immediately following the **-l** without any intervening white space. By default, libraries are located **/lib** and **/usr/lib**. If the argument is omitted, the standard system library **libc.a** is selected. Because a library is searched when its name is encountered, the placement of this option on the command line is important. The given archive library must contain valid object files as its members.

-m Produces a map or listing of the input/output sections (including holes) on the standard output.

-o *name* Names the output object file. The argument *name* is the name of the file to be used as the output file. The **default** output object file name is **a.out**. The *name* can be a full or partial system pathname.

-r Retains relocation entries in the output object file. Relocation entries must be saved if the output file is to be used as an input file in a subsequent **ld** call. When this option is used, unresolved references do not prevent the creation of an output object file.

-s Strips line number entries and symbol table information from the output object file. Relocation entries (**-r** option) are meaningless without the symbol table, hence use of **-s** precludes the use of **-r**. All symbols are stripped, including global and undefined symbols.

-ss Generates an executable aligned in 64k byte segments. The default segmentation is 1MB.

-u *sym* Introduces an unresolved external symbol into the output file's symbol table. The argument *sym* is the name of the symbol. This is useful for linking entirely from a library because initially the symbol table is empty and an unresolved reference is needed to force the linking of an initial routine from the library.

-x Does not preserve any local (non-global) symbols in the output symbol table; enters external and static symbols only. This option saves some space in the output file.

-L *dir* Changes the algorithm for searching for libraries. Looks in *dir* before looking in the default location. This option is useful for finding libraries that are not in the standard library directory. To be effective, this option must appear before the **-l** option.

-N Places the data section immediately following the text section in memory and stores the magic number 0407 in the system header. This prevents the text from being shared (the default).

-V Prints on the standard error output a " version id" identifying the link editor being run.

-VS *num* Takes *num* as a decimal version number identifying the **a.out** file that is produced. The version stamp is stored in the system header.

-xa Produces an XA/370 executable.

The following **ld** flags may be specified on AIX PS/2:

-B*num* Makes *num* the starting address for the uninitialized data (**bss**) segment of the output file. The default starting address is the first storage unit after the end of the data segment. Not all architectures support the separation of data and **bss** segments.

-d Defines common storage, even if you have specified the **-r** flag.

-D*num* Makes *num* the starting address for the initialized data segment of the output file. The default starting address begins at location

0 (if the **i** flag is in effect), at the first storage unit after the end of the text segment, or, if the **-n** flag is in effect, at the next page or segment boundary.

-e*label* Makes *label* the entry point of the executable output file.

Note: The **cc** command on AIX/370 prepends an underscore to symbol names, while AIX PS/2 does not.

-H*num* Makes *num* the boundary, usually the page size, to which the text segment must be padded if it has a different protection than does the data segment. Specify this parameter only to override the default value for the given architecture.

-i Assigns text and data segments to separate address spaces in memory, with the text segment read-only (if the architecture supports read-only memory) and shared among all users. The data segment starts at location 0 unless set with the *-D* flag. If the architecture does not support separate instruction and data space, this flag is treated as if it were the **-n** flag. (This option cannot be used with the **-K** flag.) The **-i** flag is useful primarily when you are linking executables (cross-compiling) on different machines.

-K Loads the **a.out** header into the first bytes of the text segment, followed by the text segments from the object modules. This flag causes pages of executable files to be aligned on pages in the filesystem so that they can be demand paged on systems that support paging. This flag provides mapped file support for the text and data segments.

Note: This flag is required for programs to execute on the PS/2 and is normally set by the **cc** command.

-n Makes the text segment read-only--if the architecture supports read-only memory--and shared among all users running the file. The data segment starts at the first segment boundary following the end of the text unless set with the **-D** flag. On architectures which only permit read-only text with separate text and data spaces, the **-n** flag is treated as if it were the **-i** flag. (This option cannot be used with the **-i** flag.)

-T*num* Makes *num* the starting address for the text segment of the output file. If not specified, the text segment begins at location zero.

-Z*str* Prefixes with *str* the names specified by the **-l** key. For example, with **-Z/test** and **-lxyz**, the **ld** command looks for the file **/test/lib/llbxyz.a** or, if that file does not exist, **/text/usr/lib/libxyz.a**. The ordinary directories are not searched. This flag is most useful when cross-compiling.

5.10 Link Editor Command Language

Subtopics

5.10.1 Expressions

In the AIX/370 Assembler Language, expressions may contain:

- Constant
- Global symbol
- Most of the basic C language operators

Subtopics
5.10.1.1 Constants
5.10.1.2 Global Symbols
5.10.1.3 Operators

5.10.1.1 Constants

Constants are as in C. Therefore, a number is recognized as a decimal unless it is preceded by one of the following:

0	octal
0x	hexadecimal
0X	hexadecimal

Note: 0 is the digit zero.

All numbers are treated as long integers.

5.10.1.2 Global Symbols

Symbols names may contain uppercase or lowercase letters, digits, and the underscore (_). Symbols within an expression have the value of the address of the symbol only. The link editor *does not* perform any of the following:

Symbol table lookup to find the contents of a symbo
The dimensionality of an arra
Structure elements declared in a C program

A lex-generated input scanner is used by the link editor to identify symbols, numbers, operators, and so on. Therefore, the following names are reserved and unavailable as symbol names or section names:

align	ALIGN
assign	ASSIGN
block	BLOCK
	DSECT
group	GROUP
l	
len	
length	LENGTH
	MEMORY
	NOLOAD
o	
org	
origin	ORIGIN
phy	PHY
range	RANGE
	REGION
	SECTIONS
spare	SPARE
	TV

5.10.1.3 Operators

Figure 5-1 shows the operators that are supported, in order of precedence from high to low.

Figure 5-1. Symbols and Functions of Operators
symbol
!~--(UNARYMinus)
* / %
+ -(BINARY Minus)
> > < <
= = != > < < = > =
&
|
&&
||
= + = _=*= /=

These operators have the same meaning as in the C language. Refer to *AIX/370 C Language Reference* for additional information. Operators on the same line have the same precedence.

5.10.2 Assignment Statements

External symbols may be defined and assigned addresses via the assignment statement. The syntax of the assignment statements is

```
symbol = expression;
```

or

```
symbol op= expression;
```

where op is one of the operators +, -, *, or /.

Assignment statements must be terminated by a semicolon.

All assignment statements (with the exception of the one case described in the following paragraph) are evaluated after allocation has been performed. This occurs after all input-file-defined symbols are relocated but before the actual relocation of the text and data itself. Therefore, if an assignment statement expression contains any symbol name, the address used for that symbol in the evaluation of the expression reflects the symbol address in the output object file. References within text and data (to symbols given a value through an assignment statement) access this latest assigned value. Assignment statements are processed in the same order in which they are received by the link editor.

Assignment statements are normally placed outside the scope of section-definition directives (see "Section Definition Directives" in topic 5.10.4). However, the special symbol ".", can occur only within a section-definition directive. This symbol refers to the current R address of the link editor location counter. Thus, assignment expressions involving "." are evaluated during the allocation phase of link editing. Assigning a value to the "." symbol within a section-definition directive increments/resets the link editor location counter and can create "holes" within the section, as described in "Section Definition Directives" in topic 5.10.4. Assigning the value of the "." symbol to a conventional symbol permits the final allocated address (of a particular point within the link-edit run) to be saved.

align is provided as a shorthand notation to allow alignment of a symbol to an n-byte boundary within an output section, where n is a power of 2. For example, the expression

```
align (n)
```

is equivalent to

```
(. + n - 1) &~(n - 1)
```

Link editor expressions may have either an absolute or a relocatable value. When the link editor creates a symbol through an assignment statement, the symbol's value takes on that type of expression according to the following rules:

An expression with a single relocatable symbol (and zero or mor constants or absolute symbols) is relocatable. The value is in relation to the section of the referenced symbol.

All other expressions have absolute values

5.10.3 Specifying a Memory Configuration

MEMORY directives are used to specify:

The total size of the virtual space of the target machin

The configured and non-configured areas of the virtual space

If no directives are supplied, the link editor assumes that all memory is configured. The size of the default memory is dependent upon the target machine.

By means of **MEMORY** directives, an arbitrary name of up to eight characters is assigned to a virtual address range. Output sections can then be forced to be bound to virtual addresses within specifically named memory areas. Memory names may contain uppercase or lowercase letters, digits, and the special characters $, ., or _. These memory-range names are used only by the link editor and are not carried in the output file symbol table or headers.

When **MEMORY** directives are used, all virtual memory not described in those directives is considered to be non-configured. Non-configured memory is not used in the link editor's allocation process; therefore, nothing can be link-edited, bound, or assigned to any address within non-configured memory.

As an option on the **MEMORY** directive, attributes may be associated with a named memory area. This restricts the memory areas (with specific attributes) to which an output section can be bound. The attributes assigned to output sections in this manner are recorded in the appropriate sections in the appropriate section headers in the output file. The supported attributes are:

R*:* readable memory
W*:* writable memory
X*:* executable, (instructions may reside in this memory)
I*:* initializable, (stack areas are typically not initialized).

If no attributes are specified on a **MEMORY** directive or if no **MEMORY** directives are supplied, memory areas assume the attributes of **W**, **R**, **I**, and **X**.

The syntax of the **MEMORY** directive is:

```
MEMORY
{
   name1 (attr):   origin = n1, length = n2
   name2 (attr):   origin = n3, length = n4
}
```

The keyword **origin** (or **org** or **o**) must precede the origin of a memory range, and **length** (or **len** or **l**) must precede the length as shown in the preceding example. The origin operand refers to the virtual address of the memory range. Origin and length are entered as long integer constants in either decimal, octal, or hexadecimal (standard C syntax). Origin and length specifications, as well as individual **MEMORY** directives, may be separated by white space or a comma.

By specifying **MEMORY** directives, the link editor is informed that memory is configured in some manner other than the default. For example, if it

is necessary to prevent anything from being linked to the first 0X10000 words of memory, use a **MEMORY** directive similar to the following:

```
MEMORY
{
   valid : org = 0x10000, len = 0xFE0000
}
```

5.10.4 Section Definition Directives

The purpose of the **SECTIONS** directive is to:

Describe how input sections are to be combine

Direct where to place output sections (both in relation to each othe and the entire virtual memory space)

Permit the renaming of output sections

When no **SECTIONS** directives are given (**default**), all input sections of the same name appear in an output section of that name. For example, if a number of object files from the compiler are linked, each containing the three sections

.text
.data
.bss

the output object file also contains three sections

.text
.data
.bss

If two object files are linked (one that contains sections s1 and s2 and the other containing sections s3 and s4), the output object file contains the four sections s1, s2, s3, and s4. The order of these sections would depend on the order in which the link editor sees the input files.

The basic syntax of the **SECTIONS** directive is:

```
SECTIONS
{
   secname1:
   {
      file_specifications,
      assignment_statements
   }
   secname2:
   {
      file_specifications,
      assignment_statements
   }
and so on
}
```

The various types of section-definition directives are explained in the remainder of this chapter.

5.10.5 File Specifications

Within a section-definition, the files and sections of files to be included in the output section are listed in the order in which they are to appear in the output section. Sections from an input file are specified by:

```
  filename ( secname )
```

or

```
  filename ( secnam1 secnam2... )
```

Sections of an input file are separated either by white space or commas, as are the file specifications.

If a file name appears with no sections listed, then all sections from the file are linked into the current output section. For example:

```
SECTIONS
{
   outsec1:
   {
      file1.o (sec1)
      file2.o
      file3.o (sec1,sec2)
   }
}
```

The input sections appear in the output section in the sequence given by **outsec1**:

1. Section **sec1** from file **file1.o**

2. **All** sections from **file2.0** (in the order they appear in the file)

3. Section **sec1** from file **file3.p**

4. Section **sec2** from file **file3.o**.

If there are additional input files that contain input sections also named **outsec1**, these sections are linked following the last section named in the section-definition **outsec1**. If there are any other input sections in **file1.o** or **file3.o** they will be placed in output sections with the same names as the input sections.

5.10.6 Loading a Section at a Specified Address

Binding of an output section to a specific virtual address is done by using a link editor option as shown in the following example:

```
SECTIONS
{
   outsec addr:
   {
      ...
   }
   and so on
}
```

The **addr** is the binding address expressed a a C constant. If **outsec** does not fit at **addr** (perhaps because of holes in the memory configuration or because **outsec** is too large to fit without overlapping some other output section), the link editor issues an appropriate error message.

So long as output sections do not overlap and there is enough space, they can be bound anywhere in configured memory. The **SECTIONS** directives defining output sections need not be given to the link editor in any particular order.

The link editor does not ensure that each section's size consists of an even number of bytes, or that each section starts on an even byte boundary.

However, the assembler ensures that the size (in bytes) of a section is evenly divisible by 4. You can use the link editor directives to force a section to start on an odd byte boundary. But this is not recommended because when a section starts on an odd byte boundary, the section's contents are either accessed incorrectly or are not executed properly. When you specify an odd byte boundary, the link editor issues a warning message.

5.10.7 Aligning an Output Section

You can request that an output section be bound to a virtual address that falls on an n-byte boundary (where n is a power of 2) by using the **ALIGN** option:

```
ALIGN(n)
```

is equivalent to specifying a bonding address of

```
(.+ n - 1) &~(n - 1)
```

For example,

```
SECTIONS
{
   outsec ALIGN(0x20000):
   {
      ...
   }
   and so on
}
```

The output section **outsec** is not bound to any given addresses but is linked so some virtual address that is a multiple of 0x20000 (for example, at address 0x0, 0x20000, 0x40000, 0x60000, and so on).

5.10.8 Grouping Sections Together

The link editor default allocation algorithm performs the following processes:

1. Links all input **.text** sections together into one output section. This output section is called **.text** and is bound to an address of 0x0.

2. Links all input **.data** sections together into one output section. This output section is called **.data** and is bound to an address aligned to a machine dependent constant. If the magic number (octal) is:

   0407 The text segment is not shared. The data segment is contiguous to the text segment.
   0410 The data segment begins at the next segment boundary.

   A segment in AIX/370 is 256 4k pages (0x100000). Using **ld** with the **-ss** option, a segment is 16 4k pages (0x10000).

3. Links all input **.bss** sections together into one output section. This output section is called **.bss** and is allocated so it immediately follows the output section **.data**.

   Note: The output section **.bss** is not given any particular address alignment.

If you specify a **SECTIONS** directive, this default allocation will not be performed.

The default allocation of the link editor is equivalent to supplying the following directive:

```
SECTIONS
{
   .text : { }
   GROUP ALIGN( align_value ):
   {
      .data   : { }
      .bss    : { }
   }
}
```

where align_value is a machine dependent constant; in AIX/370 a segment size is 0x100000. **GROUP** ensures that the two output sections, **.data** and **.bss**, are allocated (grouped) together. Bonding or alignment information is supplied only for the group and not for the output sections contained within the group. The sections making up the group are allocated in the order listed in the directive.

If **.text**, **.data**, and **.bss** are to be placed in the same segment, the following **SECTIONS** directive is used:

```
SECTIONS
{
   GROUP      :
   {
      .text   : { }
      .data   : { }
      .bss    : { }
   }
```

```
    }
```

Note that the three output sections (**.text**, **.data**, and **.bss**) still exist, but they are now allocated into consecutive virtual memory.

To bind to 0xC0000, use:

```
  GROUP 0xC0000 : {
```

To align to 0x10000, use

```
  GROUP ALIGN(0x10000) : {
```

With this addition, first the output section **.text** is bound at 0xC0000 (or is aligned to 0x10000); then the remaining members of the group are allocated in order of their appearance into the next available memory locations.

When the **GROUP** directive is not used, each output section is treated as an independent entity.

```
    SECTIONS
    {
       .text                : { }
       .data ALIGN(0x20000): { }
       .bss                 : { }
    }
```

The **.text** section starts at virtual address 0x0 and the **.data** section at a virtual address aligned to 0x20000. The **.bss** section follows immediately after the **.text** section if there is enough space. If there is not, it follows the**.data** section.

The order in which output sections are defined to the link editor cannot be used to force a certain allocation order in the output file.

5.10.9 Creating Holes within Output Sections

The special symbol . (dot) appears only within section definitions and assignment statements. When it appears on the left side of an assignment statement, the link editor's location counter is incremented or reset and a hole is left in the output section. Holes built into output sections in this manner take up physical space in the output file and are initialized using a fill character (either the default fill character (0x00) or a supplied fill character). See the definition of the **-f** option in "Link Editor Command Language" in topic 5.10 and the discussion of filling holes in "Initialized Section Holes or .bss Sections" in topic 5.10.12.

Consider the following section-definition:

```
Outsec:
{
     .+=0x1000;
     f1.o(.text)
     .+=0x100;
     f2.o(.text)
     . = align (4);
     f3.o (.text).
{
```

The effect of this definition is:

A 0x1000 byte hole, filled with the default fill character, is left a the beginning of the section. Input file **f1.o(.text**) is linked after this hole.

The text of input file **f2.o** begins at 0x100 bytes following the end of **f1.o(.text**).

The text **f3.o** is linked to start at the next full word boundary following the text of **f2.o** with respect to the beginning of **outsec**.

For the purposes of allocating and aligning addresses within an output section, the link editor treats the output section as if it began at address zero. As a result, in the previous example, if **outsec** is linked to start at an odd address, then the part of **outsec** built from **f3.o** (**text**) text also starts at an odd address. This occurs even though **f3.o (.text**) is aligned to a full word boundary. This is prevented by specifying an alignment factor for the entire output section.

```
outsec ALIGN(4) : {...
```

Note: The assembler always pads the sections it generates to a full word length making explicit alignment specifications unnecessary. This also holds true for the compilers.

Expressions that decrement "." are illegal. For example, subtracting a value from the location counter is not allowed because overwrites are not allowed. The most common operators in expressions that assign a value to "." are **+=** and **align**.

5.10.10 Creating and Defining Symbols at Link-Edit Time

The assignment statement of the link editor can be used to give symbols a value that is link-edit dependent. There are three types of assignments:

Use of "." to adjust link editor's location counter during allocatio (refer to "Assignment Statements" in topic 5.10.2 and "Creating Holes within Output Sections" in topic 5.10.9).

Use of "." to assign an allocation-dependent value to a symbol

Assignment of an allocation-dependent value to a symbol

Using "." to assign an allocation-dependent value to a symbol provides a means to assign addresses (known only after allocation) to symbols. For example:

```
SECTIONS
{
     outsec1: {...}
     outsec2:
     {
          file1.o (s1)
          s2_start = .;
          file2.o (s2)
          s2_end = .- 1;
     }
}
```

The symbol **s2_start** is defined to be the address of **file2.o** (s2) and **s2_end** is the address of the last byte of **file2.o** (s2).

Consider the following example:

```
SECTIONS
{
     outsec1:
     {
          file1.o (.data)
          mark = .;
          . += 4;
          file2.o (.data)
     }
}
```

In this example, the symbol **mark** is created and is equal to the address of the first byte beyond the end of **file2.o**'s **.data** section. Four bytes are reserved for a future run-time initialization of the symbol mark. The type of the symbol is integer (32 bits).

Assignment instructions involving "." must appear within **SECTIONS** definitions because they are evaluated during allocation. Assignment instructions that do not involve "." can appear within **SECTIONS** definitions but typically do not. Such instructions are evaluated after allocation is complete. Reassignment of a defined symbol to a different address is dangerous.

For example, if a symbol within **.data** is defined, initialized, and referenced within a set of object files being link-edited, the symbol table entry for that symbol is changed to reflect the new, reassigned

physical address. However, the associated initialized data is not moved to the new address. The link editor issues warning messages for each defined symbol that is being redefined within an **ifile**. However, assignments of absolute values to new symbols are safe because there are no references or initialized data associated the symbol.

5.10.11 Allocating a Section into Named Memory

It is possible to specify that a section be linked (somewhere) within a specific named memory (as previously specified on a **MEMORY** directive.) The > notation is borrowed from the redirected output concept.

For example,

```
MEMORY
{
      mem1:       o=0x000000    l=0x10000
      mem2 (RW):  o=0x020000    l=0x40000
      mem3 (RW):  o=0x070000    l=0x40000
      mem1:       o=0x120000    l=0x04000
}

SECTIONS

{
     outsec1: { f1.o(.data) } > mem1
     outsec2: { f2.o(.data) } > mem3
}
```

This directs the link editor to place **outsec1** anywhere within the memory area named **mem1** (somewhere within the address range 0x0-0FFFF or 0x120000-0x123FF). The **outsec2** is to be placed somewhere in the address range 0x70000-0xAFFFF.

5.10.12 Initialized Section Holes or .bss Sections

When holes are created within a section (as in the example in "Link Editor Command Language" in topic 5.10), the link editor normally puts bytes of zero as fill. By default, **.bss** sections are not initialized at all; that is, no initialized data (not even zeroes) is generated for any **.bss** section by the assembler nor supplied by the link editor.

Initialization options can be used in a **SECTIONS** directive to set such holes or output **.bss** sections to an arbitrary 2-byte pattern. Such initialization options apply only to **.bss** sections or holes. As an example, an application might want a non-initialized data table to be initialized to a constant value without recompiling the **.o** file or a hole in the text area to be filled with a transfer to an error routine.

Either specific areas within an output section or the entire output section may be specified as being initialized. However, because no text is generated for a non-initialized **.bss** section, if part of such a section is initialized, then the entire section is initialized. Therefore, if a **.bss** section is to be combined with a **.text** or **data** section (both of which are initialized) or if part of an output **.bss** section is to be initialized, then one of the following will be true:

Explicit initialization options must be used to initialize all **.bss** sections in the output section.

The link editor will use the default fill value to initialize all **.bss** sections in the output section.

Consider the following link editor **ifile**:

```
SECTIONS
{
     secl:
     {
        fl.o
        .=+0x200;
        f2.o (.text)
     } =0xDFFF
     sec2:
     {
        f1.o (.bss)
        f2.o (.bss) = 0x1234
     }
     sec3:
     {
        f3.o (.bss)
        ...
     } = 0xFFFF
     sec4:
     {
        f4.o (.bss)
     }
}
```

In the preceding example, the 0x200 byte hole in section **sec1** is filled with the value 0xDFFF. In section **sec2** *f1.o(.bss)* is initialized to the default fill value of 0x00, and *f2.o(.bss)* is initialized to 0x1234. All **.bss** sections within **sec3** as well as all **holes** are initialized to 0xFFFF. Section **sec4** is not initialized; that is, no data is written to the object

```
file for this section.
```

5.11 Notes and Special Considerations

Subtopics

5.11.1 Changing the Entry Point

The **a.out** header contains a field for the (primary) entry point of the file. This field is set using one of the following rules (listed in the order they are applied):

1. The value of the symbol "_start", if present, is used.

2. The value of the symbol "_main", if present, is used.

3. The value zero is used.

Thus, an explicit entry point can be assigned to this **a.out** header field by using an assignment instruction in an **ifile** of the form:

```
_start = expression;
```

If the link editor is called through **cc**, a startup routine is automatically linked in. Then, when the program is executed, the **exit** system call is called after the main routine finishes to close file descriptors and do other cleanup. Therefore, you must be careful when calling the link editor directly or when changing the entry point. You must supply the startup routine or make certain that the program always calls **exit** rather than falling through the end. Otherwise, the program will dump core.

5.11.2 Use of Archive Libraries

Each member of an archive library (such as, **libc.a)** is a complete object file and usually consists of the standard three sections:

.text
.data
.bss

Archive libraries are created through the use of the **ar** command from object files generated by running **cc** or **as**.

An archive library is always processed using selective inclusion; only those members that resolve existing undefined symbol references are taken from the library for link editing.

Libraries can be placed both inside and outside section definitions. In both cases, a member of the library is included for linking whenever:

There exists a reference to a symbol defined in that member

The reference is found by the link editor prior to the actual scannin of the library.

When a library member is included by searching the library inside a **SECTIONS** directive, all input sections from the member are included in the output section being defined. When a library member is included by searching the library outside of a **SECTIONS** directive, all input sections from the member are included into the output section with the same name. That is, the **.text** section of the member goes into **.data**, the **.bss** section of the member into **.bss**, and so on. If necessary, new output sections are defined to provide a place to put the input sections.

It is important to note the following:

1. Specific members of a library cannot be referenced explicitly in an **ifile**.

2. The default rules for the placement of members and sections cannot be overridden when they apply to archive library members.

The **-l** option is a shorthand notation for specifying an input file coming from a predefined set of directories and having a predefined name. By convention, such files are archive libraries but they do no have to be. Furthermore, archive libraries can be specified without using the **-l** option by simply giving the full or relative path name.

The ordering of archive libraries is important because to extract a member from the library it must satisfy a reference that is known to be unresolved at the time the library is searched. Archive libraries can be specified more than once and are searched every time they are encountered. Archive files have a symbol table at the beginning of the archive. The link editor will cycle through this symbol table until it has determined that it cannot resolve any more references from that library.

Consider the following example:

1. The input file **file1.o** and **file2.o** each contain a reference to the external function FCN.

2. Input **file1.o** contains a reference to symbol ABC.

3. Input **file2.o** contains a reference to symbol XYZ.

4. Library **liba.a**, member 0, contains a definition of XYZ.

5. Library **libc.a,** member 0, contains a definition of ABC.

6. Both libraries have a member 1 that defines FCN.

If you enter the **ld** command this way:

```
ld file1.o -la file2.o -lc
```

then the FCN references are satisfied by **liba.a**, member 1, ABC is obtained from **libc.a,** member 0, and XYZ is obtained from **liba.a**, member 0.

If you enter the **ld** command this way:

```
ld file1.o file2.o -lc -la
```

then the FCN references are satisfied by **libc.a,** member 1, ABC is obtained from **libc.a,** member 0, and XYZ is obtained from **liba.a, member 0.**

The **-u** option is used to force the linking of library members when the link-edit run does not contain an actual external reference to the members. For example,

```
ld -u rout1 -la
```

creates an undefined symbol called **rout1** in the link editor's global symbol table. If any member of library **liba.a** defines this symbol, that member (and perhaps other members as well) is extracted. Without the **-u** option, there would have been no trigger to cause the link editor to search the archive library.

5.11.3 Dealing with Holes in Physical Memory

When memory configurations are defined such that non-configured areas exist in the virtual memory, each application or user must assume the responsibility of forming output sections that will fit into memory. For example, assume that memory is configured as follows:

```
MEMORY
{
     mem1:        o = 0x00000         1 = 0x02000
     mem2:        o = 0x40000         1 = 0x05000
     mem3:        o = 0x20000         1 = 0x10000
}
```

The files **f1.o, f2.o, . . .fn.o** each contain the standard three sections **.text, .data,** and **.bss**. Now suppose the combined **.text** section is 0x12000 bytes. There is no configured area of memory in which this section can be placed. Appropriate directives must be supplied to break up the **.text** output section so the link editor may do allocation. For example:

```
SECTIONS
{
     txt1:
     {
            f1.o (.text)
            f2.o (.text)
            f3.o (.text)
     }
     txt2:
     {
            f4.o (.text)
            f5.o (.text)
            f6.o (.text)
      }
     etc.
}
```

5.11.4 Allocation Algorithm

An output section is formed either as a result of a **SECTIONS** directive or by combining input sections of the same name. An output section can have zero or more input sections comprising it. After the composition of an output section is determined, it must then be allocated into configured virtual memory. The link editor uses an algorithm that attempts to minimize fragmentation of memory thereby increasing the possibility that a link-edit run will be able to allocate all output sections within the specified virtual memory configuration. The algorithm proceeds as follows:

1. Any output sections for which explicit binding addresses were specified are allocated.

2. Any output sections to be included in a specific named memory are allocated. In both this and the succeeding step, each output section is placed into the first available space within the (named) memory with alignment taken into consideration.

3. Output sections not handled by one of the above steps are allocated.

If all memory is contiguous and configured (the default case), and no **SECTIONS** directives are given, then output sections are allocated in the order they appear to the link editor, normally **.text. .data, .bss**. Otherwise, output sections are allocated in the order they were defined or made known to the link editor into the first available space they fit.

5.11.5 Incremental Link Editing

As previously mentioned, the output of the link editor can be used as an input file to subsequent link-edit runs provided that the relocation information is retained (**-2** option). Large applications may find it desirable to partition their C programs into subunits, link each subunit independently, and then link edit the entire application. This is shown in the following example.

```
Step 1
     ld -r -o outfile1 ifile1

     /* ifile1 */
     SECTIONS
     {
          sul:
          {
               f1.o
               f2.o
               ...
               fn.o
          }
     }

Step 2:
     ld -r -o outfile2 ifile2

     /* ifile2 */
     SECTIONS
     {
          su2
          {
               g1.o
               g2.o
               ...
               gn.o
          }
     }

Step 3:
     ld -m -o final.out outfile1 outfile2
```

By judiciously forming subunits, applications may achieve a form of incremental link editing whereby it is necessary to re-link only a portion of the total link edit when a few programs are recompiled.

To apply this technique, there are two rules that you must adhere to:

Intermediate link edits should contain only **SECTIONS** declarations and be concerned only with the formation of output sections from input files and input sections. No binding of output sections should be done in these runs.

All allocation and memory directives, as well as any assignmen statements, are included only in the final link-edit run.

5.11.6 DSECT, COPY, and NOLOAD Sections

Sections may be given a **type** in a section definition shown in the following example:

```
SECTIONS
{    name1 0x200000 (DSECT)    : {filel.o}
     name2 0x400000 (COPY)     : {file1.o}
     names 0x600000 (NOLOAD)   : {file1.o}
}
```

The **DSECT** option creates what is called a dummy section. A dummy section has the following properties:

It does not participate in the memory allocation for output sections As a result, it does not take up any memory and does not show up in the memory map (the **-m** option) generated by the link editor.

It may overlay other output sections and even non-configured memory **DSECTs** may overlay other **DSECTs**.

The global symbols defined within the dummy section are relocate normally. That is, they appear in the output file's symbol table with the same value they would have had it the **DSECT** were actually loaded at its virtual address. **DSECT**-defined symbols may be referenced by other input sections. Undefined external symbols found within a **DSECT** cause specified archive libraries to be searched and any members which define such symbols are link edited normally (not in the **DSECT** or as a **DSECT**).

None of the section contents, relocation information, or line numbe information associated with the section is written to the output file.

In the preceding example, none of the sections from **file1.o** are allocated, but all symbols are relocated as though the sections were link edited at the specific address. Other sections could refer to any of the global symbols and they are resolved correctly.

A copy section created by the **COPY** option is similar to a dummy section. The only difference between the two is that the contents of a copy section and all associated information is written to the output file.

A section with the **type** of **NOLOAD** differs in only one respect from a normal output section:

Its text and/or data is not written to the output file

A **NOLOAD** section is allocated virtual space, appears in the memory map, and so on.

5.11.7 Output File Blocking

The **BLOCK** option (applied to any output section or **GROUP** directive) is used to direct the link editor to align a section at a specified byte offset in the output file. It has no effect on the address at which the section is allocated nor on any part of the link-edit process. It is only used to adjust the physical position of the section in the output file.

```
SECTIONS
 {
     .text BLOCK(0x200)                : { }
     .data ALIGN(0x20000) BLOCK(0x200) : { }
 }
```

With this **SECTIONS** directive, the link editor ensures that each section, **.text** and **.data** is physically written at a file offset which is a multiple of 0x200 (at an offset of 0, 0x200, 0x400,..., and so on in the file).

5.11.8 Non-relocatable Input Files

If a file produced by the link editor is intended to be used in a subsequent link-edit run, the first link-edit run must have the **-r** option set. This preserves relocation information and permits the sections of the file to be relocated by the subsequent link-edit run.

When the link editor detects an input file that does not have relocation or symbol table information, a warning message is given. Such information can be removed by the link editor (see the **-s** option in the part "Link Editor Command Language" in topic 5.10) or by the **strip** command. However, the link-edit run continues using the non-relocatable input file.

For such a link edit to be successful (to actually and correctly link edit all input files, relocate all symbols, resolve unresolved references, and so on), two conditions on the non-relocatable input files must be met:

Each input file must have no unresolved external references

Each input file must be bound to the exact same virtual address as i was bound to in the link-edit run that created it.

Note: If these two conditions are not met for all non-relocatable input files, no error messages are issued. Therefore, you must use extreme care when supplying such input files to the link editor.

5.11.9 Syntax Diagram for Input Directives

Figure 5-2. Syntax Diagram for Input Directives

Directives	**->**	**Expanded Directives**
<file>	->	{ <cmd> }
<cmd>	->	<memory>
	->	<sections>
	->	<assignment>
	->	<filename>
	->	<flags>
<memory>	->	MEMORY { <memory_spec> {[,] <memory_spec> }}
<memory_spec>	->	<name> [<attributes>] : <origin_spec> [,] <length_spec>
<attributes>	->	({ R|W |X|I })
<origin_spec>	->	<origin> = <long>
<lenth_spec>	->	<length> = <long>
<origin>	->	ORIGIN|o|org|origin
<length>	->	LENGTH|l|len|length
<sections>	->	SECTIONS { { <sec_or_group> } }
<sec_or_group>	->	<section>|<group>|<library>
<group>	->	GROUP <group_options> : { <section_list> } [<mem_spec>]
<section_list>	->	<section> { [,] <section> }
<section>	->	<name> <sec_options> : { <statement_list> } [<fill>] [<mem_spec>]
<group_options>	->	[<addr>] [<align_option>]
<sec_options>	->	[<addr>] [<align_option>] [<block_option>] [<type_option>]
<addr>	->	<long>
<align_option>	->	<align> (<long>)
<align>	->	ALIGN|align
<block_option>	->	<block> (<long>)
<block>	->	BLOCK|block
<type_option>	->	(DSECT)|(NOLOAD)|(COPY)

Syntax Diagram for Input Directives

<fill>	->	=<long>
<mem_spec>	->	> <name>
	->	> <attributes>
<statement>	->	<file_name> [(<name_list>)] [<fill>]
	->	<library>
	->	<assignment>
<name_list>	->	<name> { [,] <name> }
<library>	->	-l<name>
<assignment>	->	<lside> <assign_op> <expr> <end>
<lside>	->	<name>|.
<assign_op>	->	=|+=|-=|*=|/ =
<end>	->	;|.
<expr>	->	<expr> <binary_op> <expr>
	->	<term>
<binary_op>	->	*|/|%
	->	+|-
	->	>>|<<
	->	= =|!=|>|<|<=|>=
	->	&
	->	|
	->	&&
	->	||
<term>	->	<long>
	->	<name>
	->	<align> (<term>)
	->	(<expr)
	->	<unary_op> <term>
<unary_op>	->	!|-
<flags>	->	-e<wht_space> <name>
	->	-f<wht_space> <long>
	->	-h<wht_space> <long>
	->	-l<name>
	->	-m
	->	-o<wht_space> <name>
	->	-r
	->	-s
	->	-t
	->	-u<wht_space> <name>
	->	-z
	->	-H
	->	-F
	->	-L<pathname>
	->	-M
	->	-N

Syntax Diagram for Input Directives

	->	-S
	->	-V
	->	-VS<wht_space> <long>
	->	-a
	->	-x
<name>	->	Any valid symbol name.
<long>	->	Any valid long integer constant.
<wht_space>	->	Blanks, tabs, and new-lines.
<filename>	->	Any valid AIX/370 filename. This may include a full or partial pathname.
<pathname>	->	Any valid AIX/370 pathname (full or partial). linking ld command

6.0 Chapter 6. PS/2 Disassembler

Subtopics

6.1 CONTENTS

6.2 About This Chapter

This chapter discusses how to invoke the disassembler and how to disassemble previously compiled AIX PS/2 C, VS Pascal, or VS FORTRAN programs. This chapter also discusses the menus that can be used to define options for the disassembler. The examples shown in this chapter use AIX PS/2 C programs.

6.3 Introduction

Note: The disassembler only works with 80386 object files compiled on a PS/2. It does not read or process 370 object files.

The disassembler produces assembly language listings for programs previously compiled by AIX PS/2 C, VS Pascal, or VS FORTRAN. With the disassembler, binary code modules created by high-level languages can be translated into their assembly language equivalents, with symbolic references to external entry points.

The assembly language output may optionally include:

Absolute address listin

Hex code listin

User types listin

User variable listin

Assembly language source format accepted by the assembler (se Chapter 3, "PS/2 Assembler").

Labels indicating high-level language statement numbers

Note: These labels are not generated when compile-time optimization is used. For more information on command line options, refer to the appropriate language Users Guide.

The disassembler is flexible and easy to use. It can be executed in a variety of ways to suit your needs.

Subtopics
6.3.1 Notational Conventions
6.3.2 Preparation

6.3.1 Notational Conventions

The following notational conventions are used in this chapter:

Information that is optional is enclosed in brackets ([ ]). Bracket enclosed in quotation marks specify actual bracket characters.

Items in **bold print** must be typed exactly as shown.

Variable information that the user must supply is in ***italic***.

6.3.2 Preparation

The AIX PS/2 compilers can optionally generate additional symbolic information. To generate this symbolic information for a AIX PS/2 C program, compile the program's source code as follows:

```
cc -c d+ Cmain.c
```

This will produce a file called **Cmain.o**, which will consist of object code resulting from **Cmain.c**, plus additional symbolic information generated from **Cmain.c** as a result of the **d+** option.

6.4 Using the Disassembler

The disassembler can be executed in the following ways:

With options supplied on the command line, o

With options supplied via the menu system

The disassembler can operate on either the object file (**Cmain.o**) or the executable file (**Cmain**). Although the symbolic information generated as a result of the **d+** option is not necessary, the function of the disassembler becomes limited without it. The disassembler uses this information to:

Generate the user type declarations

Generate the user variable location information

Generate labels at the start of each user statement

Allow the user to disassemble a single entry point (as well as th entire file).

Produce unique symbol names so that the output from the disassemble can optionally be reassembled.

If an executable file is made up from two or more separately compiled units, the following situations are possible:

Each compilation unit was compiled with **d+** and each .**o** includes associated symbolic information. This is the normal case. Full symbolic information is supported for all entry points.

Not every **o** file includes associated **d+** symbolic information. This could occur when some module was not complied with **d+**. In this case, full symbolic information is limited to the entry points contained in the modules which have associated **d+** information.

Subtopics

6.4.1 From the Command Line

The disassembler is invoked from the command line using one or more options. The defaults for the disassembler are as follows:

Output is displayed to the scree

All user entry points are disassemble

The output cannot be reassemble

Address listing is generate

Hex code listing is generate

User type listing is generate

User variable listing is generate

Warning messages are printed

If you do not want to use the defaults, you must specify options.

The format for running the disassembler with options supplied on the command line is:

syntax

```
dis [options] filename[.o]
```

where:

filename[.**o**] Specifies the file to be disassembled. The file to be disassembled can be either the executable file (**filename**) or the object file (**filename.o**). A directory path may be specified with either a **filename** or a **filename[.o]** as shown below:

/usr/bin/filename

option May be any of the following:

-e entry-point name Specifies the entry point to be disassembled. An entry point is the name of a function, procedure, subroutine, program, or FORTRAN entry statement in the user's program. The entry point name is case sensitive. All Pascal and FORTRAN entry point names must be specified in lower case. If you use this option, you cannot use the **-r** option to reassemble the code.

-r The output of the disassembled program can be assembled using the Assembler. Only object files are accepted with this switch. An executable file will be rejected with the error message:

CANNOT DISASSEMBLE AN EXECUTABLE FOR REASSEMBLY

The entire file is always disassembled (the -e switch is ignored). The output does not contain:

Address listing

Hex code listing

User type listing

User variable listing.

Disassembled program instructions can be modified in any way, but disassembled data can only be modified with no addition or deletion.

You cannot use this option with the **-e** options.

This option is not available through the menu system.

-o filename Specifies the output file for disassembled code.

-w Suppresses the displaying of warning messages. This option is not available through the menu system.

Note: When the disassembler is invoked from the command line, it by default will write to standard output. Therefore, if you wish the the output to go to a file, you must redirect standard output as shown in the following example:

```
dis -r myfile.o > myfile.s
```

Or, you can use the **-o** option as shown in this example:

```
dis -r myfile.o  -o myfile.s
```

In the above examples, the commands will disassemble all the user entry points in **myfile.o** and write the output to **myfile.s**. The **myfile.s** file can be reassembled.

Subtopics
6.4.1.1 Example

6.4.1.1 Example

The following code is contained in file **Cmain.c**:

```
  void sub1 ()
  {
       printf (\%s "n", "Entered sub1");
  }

  void sub2 ()
  {
       printf (\%s "n", "Entered sub2");
  }

  int main (argc, argv)
  int argc;
  char *argv[ ];
  {
        if (*argv[1]== '1')
            sub1 ();
       else
            sub2 ();
       exit (0);
  }
```

To compile the C program contained in the file **Cmain.c**, enter:

```
  cc -c d+ Cmain.c
```

To disassemble the object file, enter:

```
  dis Cmain.o
```

The disassembled output will resemble:

```
  ENTRY POINT: Cmain.sub1

    TYPE LIST:
      NONE.

    VARIABLE LIST:
      NONE.

  sub1:
  %_sub1_1:
      0x0:   55                         pushl    %ebp
      0x1:   8b ec                      movl     %esp, %ebp
  %_sub1_2:
      0x3:   68 5c 00 00 00             pushl    $%_DohrcR
      0x8:   68 6a 00 00 00             pushl    $%_DohrcR + 0xe
      0xd:   e8 ee ff ff ff             call     printf
      0x12:  83 c4 08                   addl     $8, %esp
  %_sub1_3:
  %_sub1_exit:
      0x15:  c9                         leave
      0x16:  c3                         ret

  ENTRY POINT: Cmain.sub2
```

```
  TYPE LIST:
    NONE.

  VARIABLE LIST:
    NONE.

sub2:
%_sub2_1:
    0x18:  55                          pushl    %ebp
    0x19:  8b ec                       movl     %esp, %ebp
%_sub2_2:
    0x1b:  68 70 00 00 00              pushl    $%_DohrcR + 0x14
    0x20:  68 7e 00 00 00              pushl    $%_DohrcR + 0x22
    0x25:  e8 d6 ff ff ff              call     printf
    0x2a:  83 c4 08                    addl     $8, %esp
%_sub2_3:
%_sub2_exit:
    0x2d:  c9                          leave
    0x2e:  c3                          ret


ENTRY POINT: Cmain.main -- returns: signed 4 byte INT

  TYPE LIST:
    10 = POINTER to 8
    8 = POINTER to unsigned 1 byte CHARACTER

  VARIABLE LIST:
    argc : signed 4 byte INT
         located in %ebp+0x8
    argv : 10
         located in %ebp+0xc

main:
%_main_1:
    0x30:  55                          pushl    %ebp
    0x31:  8b ec                       movl     %esp, %ebp
%_main_2:
    0x33:  8b 45 0c                    movl     0xc(%ebp), %eax
    0x36:  8b 40 04                    movl     0x4(%eax), %eax
    0x39:  0f b6 00                    movzbl   (%eax), %eax
    0x3c:  83 f8 31                    cmpl     $49, %eax
    0x3f:  75 07                       jnz      %_main_4
%_main_3:
    0x41:  e8 ba ff ff ff              call     sub1
    0x46:  eb 05                       jmp      %_main_5
%_main_4:
    0x48:  e8 cb ff ff ff              call     sub2
%_main_5:
    0x4d:  68 00 00 00 00              pushl    $0
    0x52:  e8 a9 ff ff ff              call     exit
    0x57:  83 c4 04                    addl     $4, %esp
%_main_6:
%_main_exit:
    0x5a:  c9                          leave
    0x5b:  c3                          ret


GLOBAL's OWNER: Cmain
```

```
    TYPE LIST:
      NONE.

    VARIABLE LIST:
      NONE.
```

If the previous example was disassembled using the **-e** option, that is:

```
  dis -e sub1 Cmain.o
```

only the **sub1** subroutine would be disassembled. The disassembled output cannot be reassembled when this option is used.

Note: When the optimization option is used, statement number labels are not printed.

6.4.2 Using Menus

The disassembler can be invoked by the command:

dis

The disassembler prompts you through the possible options using a series of screens. These screens allow the user to utilize many functions of the disassembler in a simplified form.

```
+-------------------------------------------------------------------------------
¦
¦
¦
¦                    *** INPUT FILE ***
¦
¦
¦
¦
¦ Enter the name of the file you want to disassemble
¦ (or q to quit):
¦ ==>_
¦
¦
¦
¦
+-------------------------------------------------------------------------------
```

The disassembler verifies that the submitted program is a valid binary file. It then reads the corresponding **d+** information. If no corresponding **d+** information is found, the disassembler issues a warning message and prompts for the output destination. Without **d+** information, the disassembled file will have very limited symbolic capabilities and the result cannot be reassembled. When valid **d+** information is found, the next menu displayed is:

```
+-------------------------------------------------------------------------------
¦
¦
¦
¦
¦
¦
¦
¦ Do you want to reassemble the output (y or n):
¦ ==>_
¦
¦
¦
¦
+-------------------------------------------------------------------------------
```

If the input file is an object file (**.o** file), answering "yes" in the above screen causes the entire file to to be disassembled. A "no" answer will display the following menu.

```
+-------------------------------------------------------------------------------
¦
¦
¦
¦
¦                       *** ENTRY POINT ***
¦
```

```
¦ Enter the number corresponding to the entry point you want to disassemble
¦ (or q to quit).
¦
¦      0 ... ALL USER ENTRY POINTS in Cmain.o
¦      1 ... Cmain.main
¦      2 ... Cmain.sub1
¦      3 ... Cmain.sub2
¦
¦
¦ ==>_
¦
¦
¦
+-------------------------------------------------------------------------------
```

Note: Only the entry points defined in modules compiled with the **d+** flag will be displayed.

Select an Entry Point to start disassembling the submitted program. The final menu is displayed:

```
+-------------------------------------------------------------------------------
¦
¦
¦
¦
¦
¦ Enter the name of the output file, or press Enter to display the output
¦ on the screen.
¦ ==>_
¦
¦
¦
¦
+-------------------------------------------------------------------------------
```

After you have completed the disassembling, you will not be returned to the main menu screen. You can either disassemble another file, or you can enter q (Quit) to exit.

6.4.3 VS Pascal Example

The following code is contained in the file **Pmain.p**:

```
  PROGRAM example (input,output);

  PROCEDURE sub1;

    BEGIN
      WRITELN ('entered sub1');
    END;

  PROCEDURE sub2;

    BEGIN
      WRITELN ('entered sub2');
    END;

  BEGIN
    IF (argv [1]^[1] = '1')
    THEN
      sub1
    ELSE
      sub2;
  END.
```

To compile the VS Pascal program contained in the file **Pmain.p**, enter:

```
  cc d+ Pmain.p -o Pmain
```

To disassemble the executable file, enter:

```
  dis Pmain
```

The disassembled output will resemble:

```
  ENTRY POINT: example.sub1

    TYPE LIST:
      NONE.

    VARIABLE LIST:
      NONE.

  sub1:
  %_sub1_1:
      0x110: 55                       pushl    %ebp
      0x111: 8b ec                    movl     %esp, %ebp
  %_sub1_2:
      0x113: 68 fe ff ff ff           pushl    $-2
      0x118: ff 35 a4 34 40 00        pushl    0x4034a4
      0x11e: 68 c9 0d 40 00           pushl    $4197833
      0x123: e8 f4 11 00 00           call     r_openw
      0x128: 83 c4 0c                 addl     $12, %esp
      0x12b: e8 38 01 00 00           call     r_iochk
      0x130: ff 35 a4 34 40 00        pushl    0x4034a4
      0x136: 68 b8 0d 40 00           pushl    $4197816
      0x13b: 68 00 00 00 00           pushl    $0
```

```
    0x140: e8 83 01 00 00            call     r_wsv
    0x145: 83 c4 0c                  addl     $12, %esp
    0x148: e8 1b 01 00 00            call     r_iochk
    0x14d: ff 35 a4 34 40 00         pushl    0x4034a4
    0x153: e8 d4 06 00 00            call     r_wln
    0x158: 83 c4 04                  addl     $4, %esp
    0x15b: e8 08 01 00 00            call     r_iochk
%_sub1_3:
%_sub1_exit:
    0x160: c9                        leave
    0x161: c3                        ret


ENTRY POINT: example.sub2

  TYPE LIST:
    NONE.

  VARIABLE LIST:
    NONE.


sub2:
%_sub2_1:
    0x164: 55                        pushl    %ebp
    0x165: 8b ec                     movl     %esp, %ebp
%_sub2_2:
    0x167: 68 fe ff ff ff            pushl    $-2
    0x16c: ff 35 a4 34 40 00         pushl    0x4034a4
    0x172: 68 e9 0d 40 00            pushl    $4197865
    0x177: e8 a0 11 00 00            call     r_openw
    0x17c: 83 c4 0c                  addl     $12, %esp
    0x17f: e8 e4 00 00 00            call     r_iochk
    0x184: ff 35 a4 34 40 00         pushl    0x4034a4
    0x18a: 68 d8 0d 40 00            pushl    $4197848
    0x18f: 68 00 00 00 00            pushl    $0
    0x194: e8 2f 01 00 00            call     r_wsv
    0x199: 83 c4 0c                  addl     $12, %esp
    0x19c: e8 c7 00 00 00            call     r_iochk
    0x1a1: ff 35 a4 34 40 00         pushl    0x4034a4
    0x1a7: e8 80 06 00 00            call     r_wln
    0x1ac: 83 c4 04                  addl     $4, %esp
    0x1af: e8 b4 00 00 00            call     r_iochk
%_sub2_3:
%_sub2_exit:
    0x1b4: c9                        leave
    0x1b5: c3                        ret


ENTRY POINT: example

  TYPE LIST:
    NONE.

  VARIABLE LIST:
    NONE.

example:
%_example_1:
    0x1b8: e8 33 00 00 00            call     .P860421
```

```
    0x1bd: 55                          pushl    %ebp
    0x1be: 8d 2d 50 40 40 00           leal     0x404050, %ebp
%_example_2:
    0x1c4: a1 a8 34 40 00              movl     0x4034a8, %eax
    0x1c9: 8b 00                       movl     (%eax), %eax
    0x1cb: 66 0f b6 48 04              movzbw   0x4(%eax), %cx
    0x1d0: 66 83 f9 31                 cmpw     $49, %cx
    0x1d4: 75 07                       jnz      %_example_4
%_example_3:
    0x1d6: e8 35 ff ff ff              call     sub1
    0x1db: eb 05                       jmp      %_example_5
%_example_4:
    0x1dd: e8 82 ff ff ff              call     sub2
%_example_5:
%_example_exit:
    0x1e2: e8 4b 00 00 00              call     r_term
    0x1e7: 5d                          popl     %ebp
    0x1e8: e8 46 00 00 00              call     r_end
    0x1ed: c3                          ret
```

6.4.4 VS FORTRAN Example

The following code is contained in the file **Fmain.f**:

```
     subroutine test (i)

     i = i + 2
     entry xyz
     i = i**2
     return
     end

     program example

     integer month, day, year
     common /date/ month,day, year

     month = day + year
     call test(month)
     end
```

To compile the VS FORTRAN program contained in the file **Fmain.f**, enter:

```
  cc d+ Fmain.f -o Fmain
```

To disassemble the executable file, enter:

```
  dis Fmain
```

The disassembled output will resemble:

```
  ENTRY POINT: test -- returns: signed 4 byte INTEGER

    TYPE LIST:
      NONE.

    VARIABLE LIST:
      i : signed 4 byte INTEGER
           located indirectly in %ebp+0x8

  test:
      0x110: 83 ec 08                  subl     $8, %esp
      0x113: 66 c7 44 24 02 00 00      movw     $0, 0x2(%esp)
      0x11a: 8b 54 24 0c               movl     0xc(%esp), %edx
  %_test_1:
      0x11e: 8b 02                     movl     (%edx), %eax
      0x120: 83 c0 02                  addl     $2, %eax
      0x123: 89 02                     movl     %eax, (%edx)
      0x125: eb 0e                     jmp      %_test_3
  xyz:
  %_test_2:
      0x127: 83 ec 08                  subl     $8, %esp
      0x12a: 8b 54 24 0c               movl     0xc(%esp), %edx
      0x12e: 66 c7 44 24 02 01 00      movw     $1, 0x2(%esp)
  %_test_3:
      0x135: 8b 02                     movl     (%edx), %eax
      0x137: 0f af c0                  imull    %eax, %eax
      0x13a: 89 02                     movl     %eax, (%edx)
```

```
%_test_4:
%_test_exit:
    0x13c: 66 8b 44 24 02            movw     0x2(%esp), %ax
    0x141: 66 0b c0                  orw      %ax, %ax
    0x144: 74 0a                     jz       %_label_4
    0x146: 66 83 c0 ff               addw     $-1, %ax
    0x14a: 75 04                     jnz      %_label_4
    0x14c: 83 c4 08                  addl     $8, %esp
    0x14f: c3                        ret
%_label_4:
    0x150: 83 c4 08                  addl     $8, %esp
    0x153: c3                        ret



ENTRY POINT: example -- returns: 4 byte REAL*4

  TYPE LIST:
    NONE.

  VARIABLE LIST:
    day : signed 4 byte INTEGER
         located in _date_ +0x4
    month : signed 4 byte INTEGER
         located in _date_ +0x0
    year : signed 4 byte INTEGER
         located in _date_ +0x8

example:
    0x154: e8 57 2c 00 00            call     .P860421
    0x159: 55                        pushl    %ebp
    0x15a: 8b ec                     movl     %esp, %ebp
    0x15c: 68 01 00 00 00            pushl    $1
    0x161: e8 3a 09 00 00            call     f_8541
    0x166: 83 c4 04                  addl     $4, %esp
%_example_1:
    0x169: a1 5c 2d 40 00            movl     0x402d5c, %eax
    0x16e: 03 05 58 2d 40 00         addl     0x402d58, %eax
    0x174: a3 54 2d 40 00            movl     %eax, 0x402d54
%_example_2:
    0x179: 68 54 2d 40 00            pushl    $4205908
    0x17e: e8 8d ff ff ff            call     test
    0x183: 83 c4 04                  addl     $4, %esp
%_example_3:
%_example_exit:
    0x186: e8 41 0b 00 00            call     f_rtsfn
    0x18b: e8 62 2c 00 00            call     r_term
    0x190: c9                        leave
    0x191: e8 5d 2c 00 00            call     r_end
    0x196: c3                        ret
```

7.0 Chapter 7. Using the Subroutine Libraries

Subtopics

7.1 CONTENTS

7.2 About This Chapter

This chapter describes the subroutine libraries that are included with the AIX Operating System. It describes the following commonly used libraries:

C librar
Math librar
Run time services librar
Shared library

Refer to *AIX Operating System Technical Reference* for complete technical information about by function within the library.

7.3 System Libraries

The system libraries are collections of commonly used functions and declarations. You can use them in a program to avoid creating the functions for each new program. Figure 7-1 lists the system libraries.

To use the library functions:

Include any declarations for the variables that the library routine use in the program

Link the library routines with the program files after the program i compiled, or in the same process using the **cc** command.

Note: You should not store your files in the following system libraries because you could lose the files if you reinstall or update the AIX Operating System. You should instead keep your files in your own separate library.

Figure 7-1. Summary of System Libraries

Name	**Path Name**	**cc flag**	**Function**
General C Libraries:			
C library	**/lib/libc.a**	Not required	Common C language subroutines for file access, string operations, character operations, memory allocation and other functions.
Run Time Services library	**/usr/lib/librts.a**	Not Required	Support system services such as system configuration, messages, trace and error log support.
Math library	**/lib/libm.a**	**-lm**	Mathematical functions using software routines to perform floating-point arithmetic.
Programmer Workbench library	**/lib/libPW.a**	**-lPW**	Miscellaneous operating system functions.
Terminal I/O Libraries:			

curses library	**/usr/lib/libcurses.a**	**-lcurses**	Control functions for writing data to and getting data from the terminal screen.
Extended curses library	**/usr/lib/libcur.a**	**-lcur**	Control functions for writing data to and getting data from the terminal screen that support color, multiple windows, and an enhanced character set.
Other Libraries:			
Data Base Subroutine library	**/usr/lib/libdbm.a**	**-ldbm**	Data base subroutines.
Queue Backend Subroutine library	**/usr/lib/libqb.a**	**-lqb**	Subroutines for queue backends.
lex library	**/usr/lib/libl.a**	**-ll**	Subroutines for programs created by the **lex** program generator.
yacc library	**/usr/lib/liby.a**	**-ly**	Subroutines for programs created by the **yacc** program generator.
BSD compatibility library	**/usr/lib/libsd.a**	**-lbsd**	Refer to the discussion of BSD libraries in the *AIX Operating System Technical Reference*.
Graphics libraries	**/usr/lib/libplot.a** **/usr/lib/libprint.a** **/usr/lib/libgsl.a** **/usr/lib/libld.a**	**-lplot** **-lprint** **-lgsl** **-lld**	Refer to *plot* in the *AIX Technical Reference*. Also see Chapter 6 of the same book.

Subtopics
7.3.1 Including Declarations

7.3.1 Including Declarations

Some functions require a set of ***declarations*** to operate properly. You must specifically request that these declarations be included in a program. The system stores some declaration files, called ***header files***, in the **/usr/include** directory. To include a header file, use the following directive within a C language program:

```
#include <file.h>
```

where *file* is the name of one of the header files. Put all header file directives at the beginning of all files being compiled that use the header file.

7.3.2 Linking the Library Routines

When you compile a program, the **cc** program uses the **ld** program to search the C language library to locate and include functions that are used in the program. To locate and include functions from other libraries, specify these libraries on the command line when starting the **cc** command. For example, when using functions of the math library, request that the math library be searched by including the argument:

```
-lm
```

on the command line, such as:

```
cc file.c -lm
```

Use this method for all functions that are not part of the C language library. Using this method, the compiler searches the C library after searching the specified libraries. Refer to the description of the **ld** command in *AIX Operating System Commands Reference* for information about linking other libraries to a program.

7.3.3 Library Descriptions

The rest of this chapter describes the functions and header files of the libraries. Each library description begins with how to include the functions and/or header files in a program. Then, each function is listed and briefly described. Following the listing are descriptions of the header files associated with these functions (if any).

7.4 The C Library

The C library routines perform the following types of services:

Input/output contro

String manipulatio

Character manipulatio

Time function

Miscellaneous functions

The compiler loads the functions of the C library automatically. However, you must include any required declarations in the program.

Subtopics

7.4.1 Input/Output Control

The input and output (I/O) functions provide buffered I/O for a program that is easier to use than using the **read** and **write** system calls (see Chapter 8, "Using System Calls"). Do not specify any special flag to the compiler to use the I/O control functions.

Include the header file for these functions in the program. To include a header file, use the following statement:

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

All include statements should be near the beginning of the first file being compiled, usually in the declarations section before **main( )**, and must occur before using any library functions.

Subtopics
7.4.1.1 Using I/O Routines
7.4.1.2 I/O Routines Descriptions
7.4.1.3 File Access
7.4.1.4 File Status
7.4.1.5 Input
7.4.1.6 Output
7.4.1.7 Directory Access
7.4.1.8 Miscellaneous
7.4.1.9 I/O Header File

7.4.1.1 Using I/O Routines

The system treats devices as if they were files for input and output. Any of the I/O system calls or library routines can send data to or from either a device or a file. You must also open and close a device the same as a file.

Some of the I/O library routines are actually macros defined in a header file, and some are object modules of functions. In many cases the library contains a function and a macro that do the same type of operation. Consider the following points when deciding which to use:

You cannot set a breakpoint for a macro using the debug program

Functions may have side effects to avoid

Macros usually are faster than the functions because the preprocesso replaces the macros with actual lines of code in the program.

Macros result in larger object code after being compiled

Some of the I/O routines use **stdin** and **stdout** as their input and output channel. Most of the routines, however, allow you to specify a file for the source or destination of the data transfer. Some specify the file using a file pointer (which points to a structure containing the file name); others accept a file descriptor (a positive integer assigned to the file when it is opened).

Figure 7-2 summarizes some important characteristics of the input and output routines. The column headings mean:

Operation The name of the I/O routine or system call.

Macro/Function *System Call* indicates that the operation is a system call. *Function* indicates that the operation is a library function. *Macro* indicates that the operation is available as a macro, though ANSI requires that all macros be available as functions too.

Input/Output The source and/or destination of the operation is either a file that you can **Specify** or it uses **stdio** (standard input and output).

Formatted The resulting data stream is formatted (**Yes**) or not formatted (**No**).

Operation Type The data type of the information being transferred: byte, character (1-byte **char**), extended character (4-byte **wchar_t**), word (4-bytes) or string. Some conversion operations accept one type of information as input and then output a different type of information.

Figure 7-2. Comparison of I/O Operations

Operation	Macro/Function	Input/Output	Formatted	Operation Type
read	System Call	Specify	No	Byte
write	System Call	Specify	No	Byte
fread	Function	Specify	No	Byte
fwrite	Function	Specify	No	Byte

printf	Function	stdio	Yes	Byte
fprintf	Function	Specify	Yes	Byte
sprintf	Function	Specify	Yes	Byte
scanf	Function	stdio	Yes	Byte
fscanf	Function	Specify	Yes	Byte
wcprintf	Function	stdio	Yes	wchar_t
wcfprintf	Function	Specify	Yes	wchar_t
wcsprintf	Function	Specify	Yes	wchar_t
wcscanf	Function	stdio	Yes	wchar_t
wcfscanf	Function	Specify	Yes	wchar_t
ungetc	Function	Specify	No	Byte
ugetwx	Function	Specify	No	wchar_t
getc	Macro	Specify	No	Byte
getwc	Macro	Specify	No	wchar_t
getchar	Macro	stdio	No	Byte
getwchar	Macro	stdio	No	wchar_t
fgetc	Function	Specify	No	Byte
fgetwc	Function	Specify	No	wchar_t
putc	Macro	Specify	No	Byte
putwc	Macro	Specify	No	wchar_t
putchar	Macro	stdio	No	Byte
putwchar	Macro	stdio	No	wchar_t
fputc	Function	Specify	No	Byte
fputwc	Function	Specify	No	wchar_t
getw	Function	Specify	No	Word
putw	Function	Specify	No	Word
gets	Function	stdio	No	String
getws	Function	stdio	No	wchar_t string
fgets	Function	Specify	No	String
fgetws	Function	Specify	No	wchar_t string
puts	Function	stdio	No	String
putws	Function	stdio	No	wchar_t string
fputs	Function	Specify	No	String
fputws	Function	Specify	No	wchar_t string
sscanf	Function	Specify	Yes	String
wcsscanf	Function	Specify	Yes	wchar_t string

7.4.1.2 I/O Routines Descriptions

The I/O routine descriptions are grouped into the following categories:

File acces

File statu

Inpu

Outpu

Miscellaneous

In the following descriptions, ***stream*** input and output refers to sequential input and output using open file pointers. The terms ***stdin*** and ***stdout*** refer to the device or file that is currently assigned as standard input or standard output.

7.4.1.3 File Access

fclose Closes an open stream.

fdopen Associates stream with an opened file.

fileno Returns a file descriptor associated with an open stream.

fopen Opens a stream with specified permissions. A stream is what **fopen** returns.

freopen Substitutes named file in place of open stream.

fseek Repositions stream pointer.

pclose Closes a stream opened by **popen**.

popen Creates a pipe as a stream between two processes.

remove Deletes a file.

rewind Repositions stream pointer at beginning of file.

setbuf Turns buffering to stream on and off.

7.4.1.4 File Status

clearerr Resets error condition on stream.

feof Tests for end of file on stream.

ferror Tests for error condition on stream.

ftell Returns current stream pointer.

7.4.1.5 Input

fgetc Reads next character from stream (function for the macro **getc**).

fgetwc Reads next wide character from stream (function for the macro **getwc**).

fgets Reads string from stream.

fgetws Reads string of wide characters from stream.

fread Reads from stream, buffered.

fscanf Reads using format from stream.

getc Returns next character from stream.

getwc Returns next wide character from stream.

getchar Returns next character from **stdin**.

getwchar Returns next wide character from **stdin**.

gets Reads string from **stdin**.

getws Reads string of wide characters from **stdin**.

getw Reads word from stream.

scanf Reads using format from **stdin**.

sscanf Reads using format from string.

ungetc Puts one character back to stream.

ungetwc Puts one wide character back to stream.

7.4.1.6 Output

fflush Writes all currently buffered characters from stream.

fprintf Writes using format to stream.

fputc Writes next character to stream (function for the macro **putc**).

fputwc Writes next wide character to stream (function for the macro **putwc**).

fputs Writes string to stream.

fputws Writes string of wide characters to stream.

fwrite Writes to stream, buffered.

printf Writes using format to **stdout**.

putc Writes next character to stream.

putwc Writes next wide character to stream.

putchar Writes next character to **stdout**.

putwchar Writes next wide character to **stdout**.

puts Writes string to **stdout**.

putws Writes string of wide characters to **stdout**.

putw Writes word to stream.

sprintf Writes using format to string.

7.4.1.7 Directory Access

These functions provide access to directories. Use these routines to scan or read directories.

Function	Description
opendir	Open a directory for reading.
readdir	Read a directory entry.
rewinddir	Start scanning the directory again.
scandir	Review a directory.
seekdir	Seek to a location in a directory.
telldir	Find the offset in the directory.

7.4.1.8 Miscellaneous

These functions perform services that do not appear in any of the previous categories.

You do not need to specify any special option to the compiler to use these functions. There is no header file for these functions.

Function	**Description**
abort	Causes an IOT signal to be sent to the process.
abs	Returns the absolute integer value.
assert	Verifies a program assertion (place diagnostics into a program).

In addition, the following list contains miscellaneous functions:

ctermid	Returns file name for controlling terminal.
getlogin	Returns login name for owner of current process.
getuinfo	Finds the value associated with a user information name.
logname	Returns the login name of the user.
system	Executes system command.
tmpfile	Creates temporary file using node considerations.

7.4.1.9 I/O Header File

The I/O header file is **stdio.h** in the **/usr/include** directory. This file contains macro definitions and parameters that the I/O library routines use. The shell automatically opens the following files:

stdin Standard input file

stdout Standard output file

stderr Standard error file.

7.4.2 String Routines

The string routines include the following:

mbdwidth Gets display width of a multibyte character.

mblen Gets length of a multibyte character.

mbsadvance Advances pointer to next character.

mbscat Appends multibyte character string to multibyte character string.

mbschr Finds multibyte character in string.

mbscmp Compares multibyte characters in strings based on the binary ordering of the characters.

mbscpy Copies multibyte character string multibyte character string.

mbscspn Returns the length in bytes of the initial segment of the string pointed to by the s1 parameter that does NOT consist entirely of characters from the string pointed to by the s2 parameter.

mbslen Returns number of characters in string s.

mbsmbs Locates strings.

mbsncat Concatenates string pointed to by s2 to the end of string pointed to by s1. Copies a maximum of nchar multibyte characters.

mbsncmp Compares multibyte characters in strings based on the binary ordering of the characters.

mbsncpy Copies string on to string up to a maximum number of characters.

mbspbrk Locates the first occurrence in the string pointed to by s1 of any character from the string pointed to by s2.

mbsrchr Return pointer to the last occurrence of the multibyte char 'c' in string 's'.

mbsspn Returns the bytes in length of the initial segment of the string pointed to by s1 parameter that consists entirely of characters from the string pointed to by parameter s2.

mbstok Parses a string of multibyte characters into tokens.

mbstomb Converts a multibyte character into an **mbchar_t**.

mbstowcs Converts a string of multibyte chars into string of **wchar_t**.

mbtowc Converts a multibyte character into a **wchar_t**.

wcindx Returns an index value for a wide character.

wcscat Appends widecharacter string to a widecharacter string.

wcschr Searches for the first occurrence of widecharacter c in string

sp.

wcscmp Compares two widecharacter strings based on the binary ordering of the characters.

wcscpy Copies widecharacter string to widecharacter string.

wcscspn Finds the length of the initial portion of widecharacter string s1 of wide characters not in widecharacter string s2.

wcsspn Determines the number of widecharacters in a widecharacter string.

wcsncat Appends a count of widecharacters of one string to another string.

wcsncmp Compares a specific number of wide characters in one widecharacter string to another widecharacter string. The comparison is based on the binary ordering of the characters.

wcsncpy Copies a specific number of wide characters from one widecharacter string to another widecharacter string.

wcspbrk Returns 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.

wcsrchr Locates a wide character in a widecharacter string.

wcsspn Returns number of wide characters in the initial segment of a string.

wcstok 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 wide character of the text token is returned.

wcstombs Converts a string of wide chars into string of multibyte chars.

wcswcs Locates strings.

wctomb Converts a wide character into a multibyte character.

wcxcol Finds unique collating value or replacement string for character.

7.4.3 String Manipulation

The string manipulation functions include:

Locate a character position within a strin

Copy a strin

Concatenate string

Compare string

Translate a strin

Measure a string

The first group of functions require that header files be included in your program and that particular libraries be referenced. The **regexp** functions, **compile**, **step**, and **advance**, require the inclusion of header files. **regexp.h** provides standard singlebyte character functionality; **NLregexp.h** must be included for international character support. The **regcmp** and **regex** functions are in **libPW.a**.

compile Compiles a regular expression for use by **advance** and **step**. Output is encoded in a locale-independent manner.

step Using a variable set by **compile**, matches a regular expression to the beginning of a string. Supports compiling a number of regular expressions (using **compile**) before attempting a match.

advance Advances recursively to match the rest of the string to the rest of the expression, supporting metacharacters in a regular expression. The **advance** function consults the locale and uses the current collating sequence. Its use is purely for support of the **compile** and **step** functions.

regcmp Compiles a regular expression for use by **regex**. Output is encoded in a locale-independent manner.

regex Executes a compiled regular expression against a string. The locale is consulted and the current collating sequence is used.

Note that these functions are to be used in matched sets:

compile produces locale-independent output. This is processed by **step** and **advance**, which consults the locale and adjust their processing accordingly.

regcmp produces locale-independent output which is exclusively for use by **regex**. **regex** consults the locale and processes the expressions passed to it according to the current collating sequence.

All of the following functions are in **libc.a**. To use these string routines you do not need to specify any special flag to the compiler or include any particular header file in your program. If you want the compiler to check your usage, however, you should include **string.h** in your program; it contains the prototypes for these functions.

mbstowcs	Converts a string of **char**s to **whcar_t**'s.
wcstombs	Converts a string of **wchar_t**'s to **char**s.
strcat	Concatenates two strings.
wcscat	Concatenates two strings of **wchar_t**'s.
mbscat	Concatenates two strings of **char**s.
strchr	Searches string for character.
wcschr	Searches string for **wchar_t**'s.
mbschr	Searches string for **char**.
strcmp	Compares two strings using internal representations.
wcscmp	Compares two strings of **wchar_t**'s using the current collation sequence.
mbscmp	Compares two strings of **char**s using the current collation sequence.
strcpy	Copies string over string.
wcscpy	Copies **wchar_t** string over **wchar_t** string.
mbscpy	Copies string over string.
strcspn	Returns the length of initial string not containing the compared set of characters.
wcscspn	Returns the length in characters of initial string not containing the compared set of **wchar_t**'s.
NLstrcspn	Returns the length in bytes of initial string not containing the compared set of code points.
strstr	Locates a substring.
strlen	Returns the length of string.
wcslen	Returns the length in characters of string of **wchar_t**'s.
mbslen	Returns the length of string in bytes.
mbsdlen	Returns the number of display columns occupied by a string.
wcsdlen	Returns the number of display columns occupied by a string of **wchar_t**s.
strncat	Concatenates up to a specified number of bytes from one string to another string.
wsncat	Concatenates up to a specified number of **wchar_t**'s from one string to another string.
mbsncat	Concatenates up to a specified number of bytes from one string to another string.

strncmp	Compares up to a specified number of bytes from one string with another string.
wcsncmp	Compares up to a specified number of **wchar_t**'s from one string with another string.
mbsncmp	Compares up to a specified number of bytes from one string with another string.
strncpy	Copies up to a specified number of bytes from one string to another string.
wcsncpy	Copies up to a specified number of **wchar_t**'s from one string to another string.
mbsncpy	Copies up to a specified number of bytes from one string to another string.
strpbrk	Searches string for any of a set of bytes.
wcspbrk	Searches code point string for the first occurrence of any of a set of **char**s.
mbspbrk	Searches string for the first occurrence of any of a set of code points.
strrchr	Searches string backwards for character.
wcsrchr	Searches string for the last occurrence of an **wchar_t**.
mbsrchr	Searches string for the last occurrence of a code point.
strspn	Returns the length of an initial string containing a set of bytes.
wcsspn	Returns the length of an initial string of **wchar_t**'s containing the set of code points.
mbsspn	Returns the length in bytes of an initial string containing the set of code points.
strtok	Searches string for a token separated by any of a set of bytes.
wcstok	Searches string for a token separated by any of a set of wide characters.
mbstok	Searches string for a token separated by any of a set of **mbchar_t**s.
NLescstr	Converts a string possibly containing extended characters into ASCII bytes, preserving character information by converting each **NLchar** into a mnemonic string of ASCII bytes.
NLunescstr	Converts a string of ASCII bytes, possibly containing escape sequences representing extended characters, into a string in which any escape sequences are converted to **NLchar**s.
NLflatstr	Converts a character string possibly containing extended

characters into ASCII bytes, preserving character appearance by converting each code point to an ASCII character it resembles.

7.4.4 Memory Manipulation

The memory functions operate on arrays of characters in memory called ***memory areas***. The memory manipulation functions include:

Locating a character within a memory are

Copying characters between memory area

Comparing contents of memory area

Setting a memory area to a value

You do not need to specify any special flag to the compiler to use the memory functions.

Include the header file for these functions in the program. To include a header file, use the following statement:

```
#include <memory.h>
```

All include statements should be near the beginning of the first file being compiled, usually in the declarations section before **main( )**, and must occur before using any library functions.

The memory routines perform the following functions:

memccpy Copies characters from one memory area to another, stopping at the first occurrence of a specified character or after a specified number of characters are copied.

memcpy Copies a specified number of characters from one memory area to another.

memchr Finds the first occurrence of a specified character in a memory area, and returns a pointer to that character.

memcmp Compares the contents of two memory areas up to a specified maximum number of characters.

memset Sets the contents of a memory area to a specified value.

7.4.5 Character Manipulation

The character manipulation functions and macros test and translate ASCII characters.

The character manipulation functions and macros are grouped into the following categories:

Character testin

Character translatio

Character collation

Subtopics
7.4.5.1 Character Testing
7.4.5.2 Character Translation
7.4.5.3 Character Collation
7.4.5.4 Character Header File

7.4.5.1 Character Testing

Use the following functions and macros to find out the type of a character; punctuation, alphabetical and case querying functions for **wchar_t** values may vary depending on the current collation table:

isalnum Is character alphanumeric?

isalpha Is character alphabetic?

isascii Is character ASCII?

iscntrl Is character a control character?

isdigit Is character a digit?

isgraph Is character a printing character (not including space)?

islower Is character a lowercase letter?

isprint Is character a printing character (including space)?

ispunct Is character a punctuation character?

isspace Is character a white space character?

isupper Is character an uppercase character?

isxdigit Is character a hexadecimal digit?

iswalnum Is **wchar_t** alphanumeric?

iswalpha Is **wchar_t** alphabetic?

iswcntrl Is **wchar_t** a control character?

iswdigit Is **wchar_t** a digit?

iswgraph Is **wchar_t** a printing character (not including space)?

iswlower Is **wchar_t** a lowercase letter?

iswprint Is **wchar_t** a printing character (including space)?

iswpunct Is **wchar_t** a punctuation character?

iswspace Is **wchar_t** a white space character?

iswupper Is **wchar_t** an uppercase character?

iswxdigit Is **wchar_t** a hexadecimal digit?

NCisalpha Is **NLchar** an alphabetical character?

NCisupper Is **NLchar** an uppercase alphabetical character?

NCislower Is **NLchar** a lowercase alphabetical character?

NCisdigit Is **NLchar** a decimal digit?

NCisxdigit Is **NLchar** a hexadecimal digit?

NCisalnum Is **NLchar** an alphabetical character or digit?

NCisspace Is **NLchar** a space, tab, carriage return, newline, vertical tab, or form-feed character?

NCispunct Is **NLchar** a punctuation character?

NCisprint Is **NLchar** a printing character (including the space character)?

NCisgraph Is **NLchar** a printing character (excluding the space character)?

NCiscntrl Is **char** or **wchar_t** an ASCII delete character (0177) or an ordinary ASCII control character other than the single-shift characters that indicate an extended character.

wc_eqvmap Is the character an **wchar_t** that begins an (collation value) equivalence class? (Returns non-zero if it is, and 0 if it is not.) (The character must be processed by **wc_coluniq** first. See below.)

7.4.5.2 Character Translation

Use these functions to translate characters from one form to another:

toascii Converts integer to ASCII character.

tolower Converts character to lowercase.

toupper Converts character to uppercase.

wctolower Converts **wchar_t** to lowercase.

wctoupper Converts **wchar_t** to uppercase.

7.4.5.3 Character Collation

Redefinable character collation is a feature provided for international character support. Collation is performed by macros using a collation file created from a collation table by **ctab** command.

wc_collate Returns the collating value of the character for which it is called.

wc_coluniq Assigns a unique sequential value to characters in equivalence classes, so that all characters have a unique value for use by wc_eqvmap.

wc_eqvmap Returns a non-zero value if the unique collation value produced by **wc_coluniq** begins an equivalence class, which is a set of characters that can be treated as identical in some collating contexts. Otherwise, it returns 0.

7.4.5.4 Character Header File

The character header file is **ctype.h** in the **/usr/include** directory. It contains macro definitions and data declarations that the string functions use.

7.4.6 Time

The time functions access and reformat the current system date and time. You do not need to specify any special flag to the compiler to use the time functions.

Include the header file for these functions in the program. To include a header file, use the following statement:

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

All include statements should be near the beginning of the first file being compiled, usually in the declarations section before **main( )**, and must occur before using any library functions.

These functions (except **tzset**) convert a time such as the time returned by the **time** system call:

asctime Returns string representation of date and time.

ctime Returns string representation of date and time, given integer form.

difftime Computes time difference.

gmtime Returns Greenwich Mean Time.

localtime Returns local time.

tzset Sets time zone field from environment variable.

Subtopics
7.4.6.1 Time Header File

7.4.6.1 Time Header File

The header file for the time functions is **time.h** in the **/usr/include** directory. It includes declarations for variables that the time functions use, such as:

tm A structure that the **gmtime** and **localtime** functions return.

daylight An integer that is nonzero to use Daylight Savings Time conversions.

tzname A character that defines the name of time zones. The system overrides this variable if the **TZ** variables are defined in the system environment. Setting the TZ variable changes the values defined in the header file for **daylight**, **timezone** and **tzname**.

Functions that define a large number of time-related variables are also supported in the **time.h** header file. For example:

ctime Converts the structure for time (a value in seconds since 00:00:00 Greenwich Mean Time, January 1, 1970) into a character string for day, date, and time.

NLstrtime Using the structures of **ctime**, formats time and date data into strings using an international character support format established by environment variables.

NLtmtime Takes a string and sets a time structure. The string data is handled in formats established by environment variables for international character support.

7.4.7 Numerical Conversion

These functions perform numerical conversion. You do not need to specify any special flag to the compiler or include a header file to use these functions.

a64l Converts string to base 64 ASCII.

atof Converts string to floating.

atoi Converts string to integer.

atol Converts string to long.

frexp Splits floating into mantissa and exponent.

l3tol Converts 3-byte integer to long.

ltol3 Converts long to 3-byte integer.

ldexp Combines mantissa and exponent.

l64a Converts base 64 ASCII to string.

modf Splits mantissa into integer and fraction.

7.4.8 Group File Access

These functions access the group file. You do not need to specify any special flag to the compiler to use these functions.

Include the header file for these functions in the program. To include a header file, use the following statement:

```
#include <grp.h>
```

All include statements should be near the beginning of the first file being compiled, usually in the declarations section before **main( )**, and must occur before using any library functions.

endgrent Closes group file being processed.

getgrent Gets next group file entry.

getgrgid Returns next group with matching gid.

getgrnam Returns next group with matching name.

setgrent Rewinds group file being processed.

7.4.9 Password File Access

These functions search and access information stored in the password file **/etc/passwd**.

Include the header file for these functions in the program. To include a header file, use the following statement:

```
#include <pwd.h>
```

All include statements should be near the beginning of the first file being compiled, usually in the declarations section before **main( )**, and must occur before using any library functions.

endpwent Closes password file being processed.

getpw Searches password file for user ID.

getpwent Gets next password file entry.

getpwnam Returns next entry with matching name.

getpwuid Returns next entry with matching user ID.

putpwent Writes entry on stream.

setpwent Rewinds password file being accessed.

7.4.10 Parameter Access

These functions get several different types of parameters from the system. You do not need to specify any special flag to the compiler or include any header file to use these functions.

getopt Gets next option from option list in command.

getcwd Returns string representation of current directory.

getenv Returns string value associated with environment variable.

NLgetenv Returns string value associated with international character support environment variable.

getpass Reads string from terminal without echoing.

7.4.11 Hash Table Management

These functions manage hash search tables. You do not need to specify any special flag to the compiler or include any header file to use these functions.

hcreate Creates hash table.

hdestroy Destroys hash table.

hsearch Searches hash table for entry.

7.4.12 Binary Tree Management

These functions manage a binary tree. You do not need to specify any special flag to the compiler or include any header file to use these functions.

tdelete Deletes nodes from binary tree.

tsearch Searches binary tree.

twalk Walks through a binary tree to a specified level, and performs a specified action at each node of the tree.

7.4.13 Table Management

These functions manage a table. The table is a two-dimensional character array. The first subscript defines the maximum number of entries in the table. The second subscript defines the width (or length) of a single entry. These functions do not allocate storage. Be sure to allocate sufficient memory before using these functions.

You do not need to specify any special flag to the compiler or include any header file to use these functions.

bsearch Searches table using binary search.

lsearch Searches table using linear search.

qsort Sorts table using quicker-sort algorithm.

7.4.14 Memory Allocation

These functions allocate or free memory from the program.

You do not need to specify any special flag to the compiler or include any header file to use these functions.

calloc Allocates zeroed storage.

free Frees previously allocated storage.

malloc Allocates storage.

realloc Changes size of allocated storage.

7.4.15 Pseudo-random Number Generation

These functions generate pseudo-random numbers. The functions that end with **48** use a pseudo-random number generator based upon the linear congruential algorithm and 48-bit integer arithmetic. The **rand** and **srand** functions use a multiplicative congruential random number generator with period of 2(32).

You do not need to specify any special flag to the compiler or include any header file to use these functions.

drand48 Returns a random double , **n**, in the interval:

0 = n < 1.

lcong48 Sets parameters for **drand48**, **lrand48**, and **mrand48**.

lrand48 Returns a random long, **n**, in the interval:

0 = n < 2(31)

nrand48 Returns a random long, **n**, in the interval:

-2(31) = n < 2(31)

rand Returns a random integer, **n**, in the interval:

0 = n < 2(15)

seed48 Seeds the generator for **drand48**, **lrand48**, and **mrand48**.

srand Seeds the generator for rand.

srand48 Seeds the generator for **drand48**, **lrand48**, and **mrand48**.

7.4.16 Signal Handling

These functions simulate the functions available from the signal handling functions provided by the system calls for signals described in "Signal Calls" in topic 8.5.1. These functions indicate error handling to other processes, and communicate with other cooperating processes.

You do not need to specify any special flag to the compiler to use these functions.

Include the header file for these functions in the program. To include a header file, use the following statement:

```
#include <signal.h>
```

All include statements should be near the beginning of the first file being compiled, usually in the declarations section before **main( )**, and must occur before using any library functions.

These declarations define ASCII names for each of the software signals.

gsignal Sends a software signal.

raise Sends a signal.

ssignal Arranges for handling of software signals.

7.4.17 Miscellaneous

These functions perform services that do not appear in any of the previous categories.

You do not need to specify any special flag to the compiler or include any header file to use these functions.

abort Sends an IOT (I/O terminate) signal to the process.

abs Returns the absolute integer value.

ecvt Converts double to string.

fcvt Converts double to string using FORTRAN format.

gcvt Converts double to string using FORTRAN F or E format.

isatty Tests whether integer file descriptor is associated with a terminal.

labs Returns the absolute long integer value.

monitor Causes process to record a histogram of program counter location.

swab Swaps and copies bytes.

ttyname Returns the path name of terminal associated with integer file descriptor.

ttysite Finds the site number of the terminal associated with integer file descriptor.

7.5 Run Time Services Library

The run time services library routines allow you to access the following system functions from your program:

Configuration service

Message service

Trac

Error logging

The functions are in the file **/usr/lib/librts.a** which is one of the default libraries, so you can use the following command line entry to load the needed functions when linking a C language program:

```
cc file.c
```

Include header files, as needed, when using these routines. See *AIX Operating System Technical Reference* for the header files needed with each routine, as well as detailed information about their use.

cfgadev Adds a device.

cfgamni Adds a minidisk.

cfgaply Applies configuration information.

cfgcadsz Adds or replaces a stanza in an attribute file.

cfgcclsf Closes an attribute file.

cfgcdlsz Deletes a stanza from an attribute file.

cfgcopsf Opens an attribute file.

cfgcrdsz Reads an attribute file stanza.

cfgddev Deletes a device.

cfgdmni Deletes a minidisk.

errunix Logs errors that occur when running a program.

mdverify Controls write-verify operation for a minidisk.

msghelp Retrieves and displays a predefined help message.

msgimed Retrieves and outputs a predefined immediate message.

msgqued Retrieves and outputs a predefined queued message.

msgrtrv Retrieves text for a message, insert or help.

trace_on Checks whether trace channel is enabled.

trc_start Lets a process start a trace daemon.

trc_stop Lets a process stop a trace daemon.

trcunix Records trace log entries for a program.

The library also contains other routines that these routines use to perform their functions.

7.6 Math Library

The math library consists of functions and a header file. Use the following command line entry to tell the **cc** command to locate and load the needed functions when it links a C language program:

```
cc file.c -lm
```

Include the header file for these functions in the program. To include a header file, use the following statement:

```
#include <math.h>
```

All include statements should be near the beginning of the first file being compiled, usually in the declarations section before **main( )**, and must occur before using any library functions.

The math header file is **math.h** in the **/usr/include** directory. This file contains definitions for functions and macros that the math library routines use. All functions in the math library return double precision values.

The functions are grouped into the following categories:

- Trigonometric function
- Bessel function
- Hyperbolic function
- Miscellaneous functions

Subtopics
7.6.1 Trigonometry
7.6.2 Bessel
7.6.3 Hyperbolic
7.6.4 Miscellaneous

7.6.1 Trigonometry

These functions compute angles (in decimal radian measure), sines, cosines, and tangents. All of these values are expressed in double precision. The file **math.h** declares the values as double.

acos Returns arc cosine.

asin Returns arc sine.

atan Returns arc tangent.

atan2 Returns arc tangent of a ratio.

cos Returns cosine.

hypot Returns the square root of the sum of the squares of two numbers.

sin Returns sine.

tan Returns tangent.

7.6.2 Bessel

These functions calculate bessel functions of the first and second kinds of several orders for real values. The bessel functions are:

j0, j1, jn, y0, y1, and yn

For descriptions of the functions, see **bessel** in *AIX Operating System Technical Reference*.

7.6.3 Hyperbolic

These functions compute the hyperbolic sine, cosine, and tangent for real values.

cosh Returns hyperbolic cosine.

sinh Returns hyperbolic tangent.

tanh Returns hyperbolic tangent.

7.6.4 Miscellaneous

These functions do not fall into any of the previously defined categories.

ceil Returns the smallest integer not less than a given value.

exp Returns the exponential function of a given value.

fabs Returns the absolute value of a given value.

floor Returns the largest integer not greater than a given value.

fmod Returns the remainder produced by the division of two given values.

gamma Returns the natural log of gamma as a function of the absolute value of a given value.

log Returns the natural logarithm of a given value.

pow Returns the result of a given value raised to another given value.

sqrt Returns the square root of a given value.

7.7 Shared Libraries

With traditional libraries, each program has a private copy of the library code linked into the program. The AIX shared library facility lets multiple processes share the same copy of library code mapped into main memory. This facility is available on PS/2 only. The programs are linked as before, but instead of including the library code, a special section (**.lib**) is added to the program. It contains the names of the shared libraries to be used during program execution.

Shared libraries offer the following benefits:

Programs that use shared libraries may use less disk space because th library code is not linked into each executable program.

Processes that use shared libraries may require less main memor because the shared library code is only mapped into main memory once.

The system may require less time to load programs that use share libraries because the shared library code may already be in main memory.

Fewer page faults may be generated when shared libraries are use because the shared library code may already be in main memory. This can result in reduced disk activity which in turn can sometimes result in better response time.

A shared library is divided into the following two parts:

A host library fil

A target library file

The host library resembles an archive file that is used during the link edit of a program. The target library resembles an **a.out** file. It is used during the execution of a program. The AIX Operating System ensures that the required target libraries are in the address space of the executing program.

Subtopics

7.7.1 Comparing Shared and Archive Libraries

A host-shared library is similar to an archive library in several ways. Each of them are archive files. This means that they contain text and data symbols that are either defined and exported, or referenced and imported. When linking a program, the link editor searches the library for symbol definitions that resolve the program's external references. When all these references have been resolved, the program can be made executable. This is referred to as static linking.

The differences between a shared library and an archive library are how the symbolic references are resolved. To produce an **a.out** file using an archive library, the link editor copies the library into appropriate **.text** and **.data** sections in the program's object file. In contrast, to produce an **a.out** file using a shared library, the link editor does not copy any code from the library into the program's object file. Instead, it creates a special section called **.lib** in the program's object file. This section identifies the target-shared library code needed at run time to resolve the external references. When the AIX Operating System executes the resulting **a.out** file, it uses the information in the **.lib** section to load the required shared library code into memory.

To demonstrate how space is saved, consider what happens when several **a.out** files need the same code from a library. When static linking with an archive library, each program gets its own copy of the library. This results in duplication of the same code on the disk for each **a.out** file, and in memory when the programs are executed. In contrast, when a shared library is used, only the **.lib** section is added to the program, thus saving disk space. When executed, the target library is loaded in memory, where several programs can share the library code, thus saving memory.

Note: In some cases, a shared library could add space to an executable, and require more memory. When linking to a host shared library, all the library routines are included. This means that a routine that is not referenced by the program will be included, and any additional references made by that routine must be included. If a program requires a data region for a particular target library routine, it gets a copy of the data region for the entire shared library. Each process gets a private copy of those pages of the shared library's data region which are referenced by that process. Only the text section is shared among processes. Therefore, if a program only references a few library members, it may be better to link with an archive library that copies only those routines.

7.7.2 Calls to a Shared Library

Shared libraries allow you to update the shared library code without having to link the programs already using it. This is implemented by use of a branch table. For an archive library, the link editor resolves external references by binding the address of the symbol to the reference. For a shared library, each symbol is associated with an absolute address of a branch table. These addresses do not change when the shared library code is updated. An external reference is bound to that address by the link editor. Each branch table address labels a jump instruction to the actual code for the symbol. This indirect reference allows for flexibility in shared library support.

7.7.3 Using a Shared Library

Linking a file with a shared library is the same process as linking to an archive library. The only difference is the name of the library referenced. To determine whether a program has been linked to a shared library, use the **dump** command (refer to the *AIX Operating System Commands Reference Manual* for additional information on this command). The **-h** flag displays all the program section headers. If a **.lib** section is defined, then a shared library has been linked to the program. The **-L** flag displays the contents of the **.lib** section, giving you the name of the target shared library.

Using the **dbx** command, you can debug a program that has been linked to a shared library; however, you cannot debug the routines in the shared library. You may find debugging easier if you re-link your program with an archive version of the shared library.

7.7.4 Creating a Shared Library

To create a shared library, you must first create a shared library specification file. Next, you run the **shlib2** command to build the host and target libraries.

Note: This section assumes you are familiar with building an archive library.

Subtopics

7.7.4.1 The Shared Library Specification File

The shared library specification file is required by the **shlib2** command. It is a user created file that defines the necessary information to build a shared library. The following directives are defined in the shared library specification file:

#address section address
Specifies the start address of the given target file section. The start address of the **.text** and **.data** sections are defined with this directive.

The following shows the defined address ranges for AIX:

0xD0000000 - 0xD7FFFFFF	Available for Users
0xD8000000 - 0xDFFFFFFF	Reserved for AIX libraries

Note: Several shared libraries can define the same virtual addresses as long as they are not linked in the same program. Conflicts occur only within a single process, not among separate processes.

#target path
Specifies the path name of the target shared library. This is the location where the operating system looks for the shared library during execution. Normally, **path** will be an absolute path name, but it does not have to be.

Note: Only use this directive once.

#branch
Specifies the beginning of the branch table specifications. They are in the following format:

symbol position

where **symbol** is the name of an externally defined function, and **position** is the position of the **symbol's** branch table entry. **position** is an integer (i.e. 1), or range of integers (i.e. 2-10). All positions must be specified from 1 to the highest position, and no position can be duplicated.

A symbol that has more than one position associated with it, is defined as the highest position given. The other positions are considered empty slots that can be replaced by new symbols.

Note: Only use this directive once.

#objects
Specifies the names and order of the object files put into the target shared library.

Note: Only use this directive once.

#init object
Specifies that the given object file requires initialization code, as defined by initialization specifications. They are in

the following format:

pointer symbol

where **pointer** is a pointer to the imported **symbol** and must be defined in the **object** file. Each initialization sets the pointer to its default value. The generated initialization code is similar to the following format:

pointer=&symbol;

Any initialization specifications for a given object file must be defined together. Multiple specifications of the same object file are not allowed.

#ident string

Specifies a string to be included in the **.comment** section of the target library and each member of the host library.

Note: Only use this directive once.

##

Specifies a comment line.

7.7.4.2 The shlib2 Command

The **shlib2** command is used to build both the host and target libraries. This command invokes the assembler, **as**, and link editor, **ld**, as part of its processing. See the *AIX Operating System Commands Reference Manual* for more information on these commands. The syntax of the **shlib2** command is as follows:

shlib2 -s file -t target [**-h host**] [-n] [-q]

-s file Specifies the shared library specification file. This required file contains all the information necessary to create a shared library. Its contains the branch table specifications for the target, the path name in which the target should be installed, the start addresses of text and data for the target, the initialization specifications for the host, and the list of object files to be included in the shared library. See "The Shared Library Specification File" above.

-t target Specifies the name of the target shared library. When **target** is moved to the target machine, it should be installed at the location given in the specification file (see the **#target** directive in the section "The Shared Library Specification File").

Note: A new target shared library will not be generated if the **-n** option is specified.

-h host Specifies the name of the host shared library. If this option is not given, then the host shared library will not be produced.

-n Prevents a new target shared library from being generated. This option is useful when producing only a new host shared library. The **-t** option must still be supplied since a version of the target shared library is needed to build the host shared library.

-q Suppresses the printing of certain warning messages.

7.7.4.3 The shlibrpt Command

The **shlibrpt** command is a tool to help you create a shared library specification file. The input required is a list of **.o** files that are to comprise a shared library. **shlibrpt** generates the following:

A shared library specification file templat

An import.c file templat

An import.h file templat

A complete cross reference

The output of the **shlibrpt** command must be edited to create the final versions of required files for input to the **shlib2** command. The template files contain token items enclosed by angle brackets < > that need to be replaced. These include the type of a variable and the level of indirection. These two items cannot be determined from the object files. The token for the level of indirection is [*]. The rule is that the use of imported variables/routines adds one extra level of indirection. This extra level is taken care of in the template. You must fill in the original level of indirection.

The **shlibrpt** command requires that all **.o** files have a symbol table. If they are assembly files, the assembler source code must contain the .file assembly directive; otherwise, the cross reference may fail.

7.7.4.4 A Sample Shared Library

In the following example, we will build a shared library to show the procedure that you should follow, and point out certain problem areas. The example we have chosen is that of a message queue manager. The example consists of three files **queue.h**, **global.c**, and **queue.c**.

```
  /* queue.h - definition of the queue structure used in the queue manager */


  struct queue {
            struct queue *next ;
            char *qmsg ;
            } ;

  typedef struct queue QUEUE ;

  /* global.c - global variables used in the message queue manager */

  #include "queue.h"

  QUEUE *qhead = 0 ;
  QUEUE *qtail = 0 ;
  unsigned int qcount = 0 ;

  /* queue.c - queue manager example */

  #include <stdio.h>
  #include "queue.h"

  #define SUCCESS 1
  #define FAILURE 0

  /* external routines */
  extern char *calloc() ,
              *malloc() ,
              *strcpy() ;

  extern int  fprintf() ,
              strlen() ;

  extern void free() ;

  /* external variables */
QUEUEe*qhead, *qtail ;
intxqcount ;

  /* add a message to the tail of the queue */

  addq( message )
*message ;
  {
       register QUEUE *newmembr ;

       newmembr  = (QUEUE *) calloc( 1, sizeof(QUEUE) );
       if ( newmembr == NULL ) {
            fprintf( stderr, "Out of memory\n") ;
            return( FAILURE ) ;
       }
```

```
       if ( qhead == NULL ) {/* initial - no members */
            qhead =          /* update head & tail */
            qtail = newmembr ;
       }
       else {
            qtail->next = newmembr ;    /* forge link */
            qtail = newmembr ;         /* advance tail ptr */
       }

       newmembr->qmsg = malloc( strlen( message ) + 1 ) ;
       if ( newmembr->qmsg == NULL ) {
            fprintf( stderr, "Out of memory\n") ;
            return( FAILURE ) ;
       }

       ++qcount ;

       return( ( int ) strcpy(newmembr->qmsg, message) ) ;

  }

  /* remove a message from the head of the queue */

  rmq( )
  {
       register QUEUE *qcur ;

       if ( (qcur = qhead) == NULL )
            return( FAILURE ) ;     /* empty queue */

       qhead = qhead->next ;        /* new queue head */
       free( qcur->qmsg ) ;         /* free space for queue member */
       free( qcur ) ;
       --qcount ;                   /* new queue count */

       return( SUCCESS ) ;
  }

  /* flush the entire message queue */

  flushq( fdes )
  FILE *fdes ;
  {

       while ( qcount ) {
            fprintf( fdes, qhead->qmsg ) ;
            rmq() ;
       }
  }
```

Note: The **global.c** file is suggested for ease of maintenance. If any new global variables are needed as part of an update to the shared library, they can be added to the end of the old definitions. This preserves the old symbols' addresses.

To aid in the building of a shared library, we can use the **shlibrpt** command. First we compile the source files to produce the object files **global.o** and **queue.o**:

```
  cc -c queue.c global.c
```

Then use **shlibrpt** to get a report file for editing:

```
  shlibrpt queue.o global.o > REPORT
```

The report file consists of four sections:

A shared library specification fil

An import.h templat

An import.c templat

A cross reference

Note: For this small example, the cross reference is not needed and will not be shown here. However, for a large collection of files it is very useful.

After editing the report file, the three files we will be using in creating the sample shared library are shown in the following example.

```
  ## <spec filename> - specification file for shared library <name>

  #address .text <start of text>
  #address .data <start of data>

  #target <target library name>
  ## object files in library:
  #objects
             import.o
             global.o
             queue.o
  ## Branch Table Entries :
  #branch
            addq                1
            flushq              2
            rmq                 3

  ## Imported Symbols:
import.o
             _<lib name>_ _iob
             _<lib name>_calloc             calloc
             _<lib name>_fprintf            fprintf
             _<lib name>_free               free
             _<lib name>_malloc             malloc
             _<lib name>_strcpy             strcpy
             _<lib name>_strlen             strlen

  ## ** END OF SPEC FILE TEMPLATE **


  /* import.h - template for macro re-definitions */

  #define _iob    (*_<lib name>_ _iob)
  #define calloc  (*_<lib name>_calloc)
  #define fprintf (*_<lib name>_fprintf)
  #define free    (*_<lib name>_free)
```

```
#define malloc  (*_<lib name>_malloc)
#define strcpy  (*_<lib name>_strcpy)
#define strlen  (*_<lib name>_strlen)

extern <type> [*]_iob();
extern <type> [*]calloc();
extern <type> [*]fprintf();
extern <type> [*]free();
extern <type> [*]malloc();
extern <type> [*]strcpy();
extern <type> [*]strlen();

/*** END OF IMPORT.H FILE TEMPLATE ***/
/* import.c - template for imported symbol initialization */

<type>     [*](*_<lib name>_ _iob)() = 0 ;
<type>     [*](*_<lib name>_calloc)() = 0 ;
<type>     [*](*_<lib name>_fprintf)() = 0 ;
<type>     [*](*_<lib name>_free)() = 0 ;
<type>     [*](*_<lib name>_malloc)() = 0 ;
<type>     [*](*_<lib name>_strcpy)() = 0 ;
<type>     [*](*_<lib name>_strlen)() = 0 ;

/*** END OF IMPORT.C FILE TEMPLATE ***/
```

Further editing is needed to get these three files into the final form needed to build a shared library. Beginning with the specification file, we must choose the addresses for the library's **.text** and **.data** sections. From the segments reserved for private use we choose the following:

```
text:   0xD0000000
data:   0xD0400000
```

Note: These region addresses must be on a page table boundary, which is 0x400000 for AIX on the PS/2.

Next, we choose the full path name for our target shared library. In this example, we will choose the current directory with the library name of **libq_s** so the full path name for the target is **./libq_s**.

Finally, we replace the <lib name> tokens in the **#init** directive of the specification file with our chosen library name, **libq**. Using the library name as a prefix to the imported symbol pointers reduces the chance of a name conflict occurring. With these changes, the final specification file appears as shown below:

Note: In the list of object files (**#objects**), **import.o** is first. This is because imported data files should be defined first to preserve future compatibility. Otherwise, a change in the size of static data in **queue.o** would change the addresses of the external data symbols.

```
## libq.sl - specification file for shared library libq_s

#address .text 0xD0000000
#address .data 0xD0400000

#target ./libq_s

## object files in library:
```

```
  #objects
            import.o
            global.o
            queue.o
  ## Branch Table Entries :
  #branch
           addq                1
           flushq              2
           rmq                 3

  ## Imported Symbols:
import.o
            _libq_ _iob
            _libq_calloc              calloc
            _libq_fprintf             fprintf
            _libq_free                free
            _libq_malloc              malloc
            _libq_strcpy              strcpy
            _libq_strlen              strlen
```

Since it is convenient to have a common source base for both shared and unshared versions of a library, we have the **import.h** file. We change the original source code to add a conditional statement that includes the **import.h** file when a shared library version is to be made.

Following the same procedure as in the specification file, we will replace all occurrences of the token <lib name> with the library name **libq**. The **shlibrpt** command analyzed the object files of the library to produce the **import.h** file. Since there is no information in the object file that indicates the type of an imported variable, we must check the source file or AIX documentation to determine the type. We then replace the <type> token with the appropriate type of the imported variable. Next, we must determine the level of indirection of the imported variable. For example, the routine **calloc()** returns a pointer to a char, so in the **import.h file**, we have the declaration:

```
  extern char *calloc() ;
```

The **shlibrpt** command also cannot determine if an external reference is made to a function or to a variable in the **.data** section. Therefore, **shlibrtp** assumes that all references external to the shared library are to external functions. This is true for the majority of external references. In our example; however, we have one exception to this rule. The variable **_iob** is actually a reference to an array defined in **stdio.h**. This makes the declaration in **import.h** redundant. Therefore, in the edited file **import.h** which appears below, we have removed the extern declaration for **_iob**.

```
  /* import.h - macro re-definitions for libq */

  #define    _iob   (*_libq_ _iob)
  #define    calloc   (*_libq_calloc)
  #define    fprintf  (*_libq_fprintf)
  #define    free     (*_libq_free)
  #define    malloc   (*_libq_malloc)
  #define    strcpy   (*_libq_strcpy)
  #define    strlen   (*_libq_strlen)

  extern char *calloc();
  extern int fprintf();
```

```
extern void free();
extern char *malloc();
extern char *strcpy();
extern int strlen();
```

Finally, we need a source file to hold the definitions of the imported symbol pointers. This is the purpose of the **import.c** file which appears in edited form below. Note the correction to the declaration of the pointer to **_iob**. Also note the inclusion of **stdio.h**. This is necessary since the type FILE is defined in this include file.

```
/* import.c - imported symbol initialization for libq */

#include <stdio.h>

FILE  (*_libq_ _ iob)[]= 0 ;
char *(*_libq_calloc)() = 0 ;
int  (*_libq_fprintf)() = 0 ;
void (*_libq_free)() = 0 ;
char *(*_libq_malloc)() = 0 ;
char *(*_libq_strcpy)() = 0 ;
int  (*_libq_strlen)() = 0 ;
```

A fragment of the modified source file **queue.c** appears below. For this example, the only difference between it and the original is the inclusion of the new include file **import.h** which **must** be included before **stdio.h**.

```
/* queue.c - modified queue manager example */

#include "import.h"

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

Now we are ready to build the sample shared library. First, we will compile the source files:

```
cc -c import.c global.c queue.c
```

Then we use **shlib2** to build both our host and target libraries:

```
shlib2 -s libq.sl -t libq_s -h libq_s.a
```

Presuming all of the source files compile correctly, the **shlib2** command line shown above will create both the host library, **libq_s.a**, and the target library, **libq_s**.

In order to test the correct operation of the **libq** shared library, we created the following test program:

```
/* qtest.c - test program using example lib libq */

#include <stdio.h>

main()
{
      char msgbuf[256];
```

```
        register int i ;

        addq( "This is test 0\n" ) ;
        flushq( stderr ) ;

        for ( i=1 ; i<11 ; ++i ) "
             sprintf( msgbuf, "This is test %d\n", i ) ;
             addq( msgbuf ) ;
        }

        flushq( stderr ) ;

  }
```

The following command compiles and links the test program with the shared library, giving it the name **qtest**.

```
  cc -o qtest qtest.c libq_s.a
```

Finally we run the test program by using the command:

```
  /qtest
```

7.7.4.5 Guidelines

The guidelines described here stress ways to increase sharing code while avoiding the disadvantages of a shared library. The guidelines also stress upward compatibility.

You should keep in mind the following restrictions to building a shared library. These restrictions involve static linking.

Exported symbols have fixed addresses. Therefore, if an exporte symbol moves, you have to re-link all programs that use the shared library and data.

If the library's text changes for one process at run time, it change for all processes. Only the data can change because each process has a private copy of the shared libraries data section.

If the library uses a symbol directly, that symbol's run time valu (address) must be known when the library is built.

Imported symbols cannot be referenced directly. Their addresses ar not known when you build the library, and they can be different for different processes. You can use imported symbols by adding an indirection through a pointer in the library's data.

7.7.4.6 Choosing Library Members

The members of a shared library determine the efficiency of the library. Each routine should be considered to see if it is best used in a shared or archive library. The following guidelines may help you in creating the list of objects for a shared library:

Include large, frequently used routine

Exclude infrequently used routine

Exclude routines that use too much static dat

Exclude routines that complicate maintenanc

Include routines the library itself needs

The large, frequently used routines are best used in a shared library. They save disk space for individual programs, and save memory when concurrent processes share the same code.

Infrequently referenced routines in a shared library can degrade performance, especially on paging systems. Because all the routines in a shared library are loaded in memory, a page fault may be more likely to occur if a called function is surrounded by unrelated library code. If a shared library has unrelated functions, and unrelated processes make random calls to those functions, the locality of reference may be decreased, causing more paging activity.

Routines that define static data increase the size of processes. Each process that uses a shared library gets its own private copy of all the library's data, regardless of how much of the data is actually needed. Therefore, you should not include routines that require a lot of static data. You can import global data, if necessary, but not local, static data.

All exported symbols must remain at constant addresses. The branch table makes this easy for text symbols, but data symbols do not have an equivalent mechanism. The more data a library has, the more likely some of them will have to change size. Any change in the size of exported data may affect symbol addresses and break compatibility. Avoid such routines as they complicate maintenance.

You may wish to consider including all the routines referenced by existing library routines. This would make the shared library self-contained. You should, however, take into consideration the previously mentioned guidelines first.

7.7.4.7 Writing Code to a Shared Library

Code written to work in a shared library will also work in an archive library. However, the reverse is not true because a shared library must explicitly handle imported symbols. Code should be written to be compatible between the two types of libraries, and to be compatible between different versions of shared libraries.

Note: An archive version may be slightly bigger and slower.

The following guidelines will help you attain this goal:

Minimize global dat

Define text and global data in separate source file

Initialize global dat

Preserve branch table order

In a shared library, all external data symbols are global; meaning they are visible to applications. This can complicate maintenance because the addresses of external variables must remain constant. Therefore, you should try to reduce the use of global data.

One way to reduce the use of global data is to use automatic (stack) variables instead of permanent storage. You can also use static data, which is not visible outside the library.

Data buffers can be allocated at run time instead of defined at compile time. This reduces the size of the library's data region for all processes, and saves memory so that only the processes that actually need the buffers get them. It also allows the size of the buffer to change without affecting compatibility. Statically allocated buffers cannot change size without affecting the addresses of other symbols and, perhaps, breaking compatibility.

Static data can affect compatibility when mixed with global data. When the size of static data changes, it shifts the addresses of any global variables declared after it. This can be within the same object file, or in any object files linked after it.

To prevent data symbol addresses from moving, you can separate text from global data. If new exported variables are needed, they can be added at the end of the old definitions to preserve the old symbols addresses. It helps to group all exported data symbols and place them at lower addresses than the static data. This is controlled by the order of the list of object files in the shared library specification file. Place the object files containing the global data definitions at at the top of the list.

Note: There is no penalty for a shared library to define all globals in a single file. There is however, an impact to an archive library version using the same code.

Initializing variables is another way to prevent their address from changing. The order of uninitialized variables may not be consistent from one link edit to the next. However, the link editor will not change the order of the initialized variables, thus allowing a library developer to preserve compatibility. Therefore, you should initialize exported variables, including the pointers for imported symbols. Even though this

uses more disk space in the target shared library, the expansion is limited to a single file.

The order of the specification file branch table should be maintained with previous versions of the shared library. You should add any new function only at the end of the branch table. As long as previous assignments are not changed, existing programs do not have to be re-linked with the new version of the shared library.

7.7.4.8 Importing Symbols

Symbols defined outside a shared library cannot be directly accessed by the shared library routines. However, a mechanism exists where you can define pointers in the data area and have them initialized to the address of the imported symbols. This allows you to access imported symbols indirectly, delaying symbol binding until run time. Both text and data symbols can be imported. A shared library cannot contain unresolved references. Therefore, any undefined symbols must be imported. The imported symbols can come from the user's code, another library, or even the shared library itself.

Having a shared library import its own symbols is useful if you wish to allow redefinition of these symbols. A routine can be re-defined outside the shared library and still be used by other routines within the shared library. This provides full compatibility between a shared and equivalent archive library.

Shared library code is different from archive library code because of importing symbols. The references to external symbols would change to become an indirect reference to the import symbol pointers. This difference in the source code can be hidden through the use of macro definitions. You could define different versions of a header file; one for a shared library that contains the macro definitions, and one for an archive library that would be empty. The **-I** flag to the **cpp** command would specify the appropriate directory from which to get the header file. Another way to keep common source code is to only define the shared library header file, and conditionally include it in the source code. In this case, you would use the **-D** flag to **cpp** to define the conditional variable when you make the shared library version. To complete the transformation to shared library code, the pointer to the import symbol must be defined and initialized to zero. This can be done in a separate C source file to maintain source compatibility. "A Sample Shared Library" in topic 7.7.4.4 shows you the mechanics of importing symbols.

All imported symbol pointers should have initialization code defined in the specification file. **shlib2** adds relocatable code that performs the import initialization to the specified object file in the host shared library. It creates an unresolved reference to make sure the symbol being imported gets resolved. When the link editor extracts the object file from the host library, the relocatable code goes into the executable file. It resolves the unresolved references and collects all initialization code. When the file is executed, the system startup routine executes the initialization code which sets the library pointers to their appropriate values. This is done before the process calls **main**.

You may wish to consider writing C source files that define imported symbol pointers singly or in related groups to reduce unnecessary loading. This will give the link editor a finer granularity to use when it resolves the symbols. Otherwise, the link editor may resolve symbols that are not required.

7.7.4.9 Tuning Shared Library Code

Shared library code can be tuned by grouping routines so they share the same page in memory. To get information on the static and dynamic calling dependencies, you can use some system commands.

With the **prof** command, you can profile an archive version of the library to decide which routines should be included in the shared library. Note that the program size is a static property, and paging is a dynamic property. These static and dynamic characteristics may conflict, so you have to decide whether the performance lost is worth the disk space gained.

You can improve the locality of references by grouping dynamically related functions. The **cflow** command generates this static dependency information. Combine it with profiling to see what things actually are called, as opposed to what things might be called.

You can also arrange the shared library target's object files so that frequently used functions do not cross page boundaries unnecessarily. Using the **dis** (disassembler) command, you can look at the code, grouping related function and then dividing the them into page size chunks. When arranging object files within the target library, be sure to keep the text and data files separate. You can reorder text object files without breaking compatibility; the same is not true for object files that define global data. The infrequently called functions can be used as glue between the chunks. Because the glue between pages is referenced less frequently than the page contents, the probability of a page fault is decreased.

Finally, you may have to consider the hardware you are using to obtain better performance. You get better performance by arranging the typical process to avoid cache entry conflicts. If a heavily used library had both its text and its data segment mapped to the same cache entry, the performance penalty would be particularly severe. Every library instruction would bring the text segment information into the cache. Instructions that referenced data would flush the entry to load the data segment. Of course, the next instruction would reference text and flush the cache entry again.

7.7.4.10 Archive and Shared Library Compatibility

It is suggested that you create both compatible archive and shared libraries. This allows you to easily substitute one for the other. For various reasons, you may not wish to include all the routines from an archive library in the target shared library version. Compatibility can still be maintained by first building the host and target library to include only the selected routines. Then you can add the missing relocatable objects to the host library. See the **-n** flag of the **shlib2** command.

7.7.4.11 Shared Library Upward Compatibility

If a shared library is updated and compatibility is not maintained, all the programs that were linked with the old shared library must be re-linked with the new version. If they are not re-linked, these programs will not execute properly.

Libraries are incompatible if their exported symbols have different addresses. Shared library developers normally want newer versions One way to compare the exported symbol addresses is by using the **nm** command and a **sed** script. The **nm** command displays the symbol table information and the **sed** script filters out unnecessary information. Only the external symbol names and addresses are required for comparison. The output for the old and new shared libraries is compared using the **diff** command. No differences indicated compatibility of the exported symbols. See the *AIX Operating System Commands Reference Manual* for information on **nm**, **sed**, and **diff**.

sed command file:

```
/|extern|.*/!d
s///
/^.bt/d
/^etext /d
/^edata /d
/^end /d
```

Note: A library can also be incompatible if the number of arguments to a function changes. A change of this sort is not reflected by the above process.

Command lines to execute comparison:

nm oldsl ¦ **sed -f** cmdfile **>**old.nm

nm newsl ¦ **sed -f** cmdfile **>**new.nm

diff old.nm new.nm

If two versions of a shared library are incompatible and it is not possible to re-link all the programs, then a new target path name can be used. Since the host and target path names are independent, you do not have to change the host library path name. New programs being linked will use the new target library. Old programs will continue to access the old target library.

Note: Maintaining multiple library versions may use more disk space and memory than the equivalent archive library.

8.0 Chapter 8. Using System Calls

Subtopics
8.1 CONTENTS
8.2 About This Chapter
8.3 Header Files Needed for Calls
8.4 Process Calls
8.5 Interprocess Communications
8.6 File System Calls
8.7 Time System Calls

8.1 CONTENTS

8.2 About This Chapter

This chapter discusses how to access the services of the operating system from a program through system calls. It also contains information on process calls, interprocess communications, file system calls, and time system calls. See *AIX Operating System Technical Reference* for reference information about a system call described in this chapter.

8.3 Header Files Needed for Calls

Some ***system calls*** depend on special macro definitions and declarations for the values that they return. The system provides this information in files called header files that are in the directory **/usr/include**. Therefore, when using system calls, be sure to include any header files that the system call needs. To include a file in a program, use the following statement in the program:

```
#include <file.h>
```

where the parameter, *file.h*, represents the name of the header file to use.

The header files needed for each system call are:

Header File	Calls That Use The Header File
fcntl.h	open, fcntl
grp.h	getgroups, setgroups
mon.h	profil
signal.h	kill, sigaction, sigprocmask
uinfo.h	usrinfo
unistd.h	access, utime
ustat.h	ustat
sys/chownx.h	chownx
sys/devinfo.h	ioctl
sys/ioctl.h	ioctl
sys/ipc.h	msgxrcv, msgctl, msgget, msgsnd, msgrcv, semctl, semget, semop, shmat, shmctl, shmdt, shmget
sys/lockf.h	lockf
sys/lock.h	plock
sys/msg.h	msgxrcv, msgctl, msgget, msgsnd, msgrcv
sys/ptrace.h	ptrace
sys/select.h	select
sys/sem.h	semctl, semget, semop
sys/shm.h	disclaim, shmctl, shmget, shmat, shmdt
sys/dustat.h	dustat
sys/stat.h	stat, fstat, statx, fstatx, creat, chmod, msgget, mknod, shmget, semget

sys/times.h	times
sys/types.h	Most system calls including but not limited to fstat, fstatx, seek, msgxrcv, msgctl, msgget, msgrcv, msgsnd, semctl, semget, semop, shmat, shmctl, shmdt, shmget, stat, statx, times, ulimit, ustat
sys/utsname.h	uname
sys/wait.h	wait, wait3

8.4 Process Calls

When a program runs in the system, that program, together with the environment that it runs in, is a ***process*** Many processes are running in the system: some running system programs (like **init**) and some running application programs. When each process begins, the system assigns it an identification number (***process ID***) that is a positive integer. Except for certain system processes, the process ID of a process is the site number multiplied by 100,000 plus an integer between 0 and 99,999. For example, site 25 has process IDs between 2,500,000 and 2,599,999. As long as the process remains active, the system uses this number to identify that process. When the process ends, the system can assign the number to a new process.

Subtopics

8.4.1 Process Handling Calls

Use the following calls to control creating, operating and stopping processes (see *AIX Operating System Technical Reference* for more information about these calls):

Call	Description
brk, sbrk	Changes data segment space allocation.
exec	Runs a new program in the currently running process.
exit	Causes a process to finish execution.
fork	Creates a new process.
migrate	Moves a process to another cluster site.
getpriority	Changes or determines priority.
setpriority	Changes or determines priority.
pipe	Creates an inter-process channel.
plock	Locks process, text, or data in memory.
profil	Starts and stops execution profiling.
ptrace	Traces the execution of another process.
rexec	Runs a program at a specified cluster site.
rfork	Creates a new process at a specified cluster site.
run	Creates a new process and runs a program at a specified cluster site.
wait, wait3, waitpid	Waits for a child process to stop or terminate.

8.4.2 Starting a Process

A new process can be started only with one of the system calls **fork**, **rfork**, or **run**. The other system calls described below are used to control process execution.

Subtopics

8.4.2.1 Fork

To split the currently running process into two that execute independently, the **fork** call does the following:

1. Gets a new process id from the system.

2. Places a copy of the current image in the new process space.

3. Returns to the **fork** call with a 0 in child and process id in parent.

The original process called the parent, and the new process called the child have independent copies of the original process data image but share all open files. If the parent process was running a shared text load module, a single copy of the text will be shared by both the parent and child.

The two processes differ only in the values returned by the **fork** call. The value returned to the parent process is either the nonzero process id of the newly created child, or -1 if an error occurred. The child process always receives a value of zero.

Use the following conditional statement to determine whether the process making the call is the parent or child:

```
int status;
     if((procid = fork()) == 0)
     {
          execv(path, arguments);    /* executed by child  */
     }
     wait(&status)              /* executed by parent */
```

8.4.2.2 Rfork

The **rfork** system call is an extended version of the **fork** system call. **Rfork** takes a single argument, which is the number of the cluster site where the child is to be created. If the local cluster site number is specified, it is equivalent to **fork**. The use of **rfork** in the following conditional statement lets you specify the site on which the new process is to be created:

```
int status;
     if((procid = rfork(site_number)) == 0)
     {
          execv(path, arguments);
     }
     wait(&status);
```

8.4.2.3 Exec

If the **fork** or **rfork** call is successful, the next system call that the child process executes is usually an **exec** system call or one of its derivatives. This call replaces the program currently running in the child process with a new program and starts executing it. One form of this call, shown in the previous conditional statement, is **execv(path, arguments)**. All forms of **exec** replace the text and data of the calling process with the contents of the file *path* but leave unaltered the status of open files and current working directory. The new program is then executed with the specified arguments.

If the file specified by *path* cannot be executed on the cluster site of the process performing the **exec** system call, then the system will look for a cluster site that can. Refer to the discussion of **getspath, setspath** in the *AIX Operating System Technical Reference*.

8.4.2.4 Rexec

The **rexec** system call is an extended version of the **exec** system call. **Rexec** takes a single argument, which is the number of the cluster site where the child is to be created. If the local cluster site number is specified, it is equivalent to **exec**.

```
if((procid = fork()) == 0)
{
     rexecv(path, arguments, site_number);
}
wait(&status);
```

8.4.2.5 Run

This call behaves like a **fork** call followed by a call to **rexec** but executes much more efficiently. A new process is created on the indicated site and then overlaid by the contents of the specified file with control transferred to the new entry point. **Run** does not return a value to the child, so the construction is now slightly different:

```
 if((procid = runv(path, arguments, site_number, fdmapsize, fdmap)) != -1)
{
   wait(&status);
}
```

8.4.2.6 Migrate

The call **migrate (site_number)** moves the calling process to the specified site, if possible. Migration (and, in general, remote tasking within the cluster) is not permitted under certain conditions, such as when the calling process uses semaphores, messages, or shared memory.

8.4.2.7 Wait

The **wait** system call suspends a parent process until any of its child processes complete execution. **Wait** returns the child process ID. The exit status of the child process is stored in the status parameter.

8.4.2.8 Example of Process Life Cycle

When the shell executes a command, it performs the fork and wait process just described. See Figure 8-1 for an illustration of the following process. For example, the command:

```
pr
```

tells the shell to execute a program called **pr**. To do this:

1. The shell issues a **fork** system call to get a new process.

2. The system marks the shell data and stack space for the new process space in such a way as to avoid unnecessary system overhead. On the PS/2 (386), the system marks this information as ***copy on write***. On the System/370, the system marks this information as ***copy on reference***.

3. The system schedules the new process to be run.

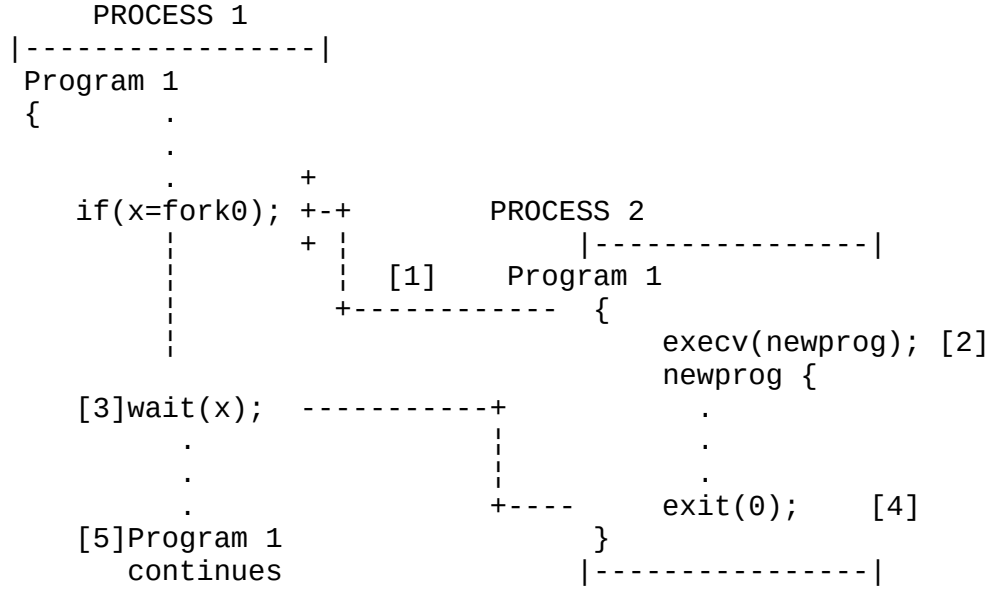
4. The first shell issues a **wait** to wait for the new process (child) to complete.

The new process (also a shell) then:

1. Issues an **exec** system call to load and run the **pr** program in the new process.

2. Issues an **exit** system call with a completion code when it is done.

When the parent shell process receives the completion code from the **exit** system call in the child, it starts running again. You can then enter another command on the command line.



---

[1] Process 1 forks to get Process 2.
[2] Process 2 executes next instruction in Program 1

```
    which copies newprog into Process 2.
[3] Process 1 executes a wait() call.
[4] Newprog completes with an exit() call, which
    returns a value to Process 1; Process 2 stops.
[5] Program 1 starts running again.
```

Figure 8-1. Using the Fork System Call

8.4.2.9 Special Processes

Special processes have process IDs less than 100,000. The special process **INIT** (the initialization process, which has a process ID of 1) is initially created to initialize the system and is the ancestor process to all others in the system. Other special processes swap processes out to disk, maintain a minimum amount of free memory on the system free page list, service network requests, manage the dynamic inclusion and exclusion of cluster sites, and propagate committed updates for replicated files.

8.4.2.10 Example of Fork and Wait System Calls

The program in Figure 8-2 uses the **fork** and **wait** system calls in a simple program that runs. Compile the program using the **cc** command:

```
cc -o forktst2 forktst2.c
```

To run the program, enter the command:

```
forktst2
```

When the program runs, it writes output to the screen. The two processes that the program creates both write to the screen, but this time the parent waits for the child to complete before it writes to the screen. Therefore, the result that appears on the screen is much more predictable than the previous example. The output looks like Figure 8-3.

```
#include <stdio.h>
#define PR(value)  fprintf(stdout,"%s\n",(value) )
#define PD(value)  fprintf(stdout,"%d\n",(value) )

main()
{
  int c_count;
  int p_count;
  int status;
  int frkpid;
  int chldpd;

  PR("starting main process");
  if( (frkpid = fork() ) == 0)
  {
     PR("starting child process");
     for(c_count = 0; c_count <= 4; c_count++)
        PR("child");
     PR("ending child process");
     PD(frkpid);
     exit(0);
  }
  chldpd = wait(&status);
  PR("starting parent process");
  for(p_count = 0; p_count <= 4; p_count++)
     PR("parent");
  PR("ending parent process");
  PD(frkpid);
  PD(chldpd);
  PD(status);
}
```

Figure 8-2. Fork and Wait System Calls - Sample Program

```
starting main process
starting child process
child
```

```
child
child
child
child
ending child process
0
starting parent process
parent
parent
parent
parent
parent
ending parent process
100096
100096
0
$
```

Figure 8-3. Output from forktst2 Sample Program

8.4.2.11 Example of Exec System Call

The program in Figure 8-4 uses **exec** system call together with the **fork** and **wait** system calls in a simple program that runs. Compile the program using the **cc** command and copy it to its own file, **forktst3**, as in the previous **fork** example.

To run the program, enter the command:

```
forktst3
```

When the program runs, it writes output to the screen. The two processes that the program creates both write to the screen; the parent waits for the child to complete before it writes to the screen. The output looks like Figure 8-5.

```
#include <stdio.h>
#define PR(value)  fprintf(stdout,"%s\n",(value) )

main()
{
  int p_count;
  int status;
  int frkpid;
  int chldpd;

  PR("starting main process");
  if( (frkpid = fork() ) == 0)
  {
     PR("starting child process");
     execl("/bin/"date", "date", (char*)0);
  }
  chldpd = wait(&status);
  PR("starting parent process");
  for(p_count = 0; p_count <= 4; p_count++)
     PR("parent");
  PR("ending parent process");
}
```

Figure 8-4. Exec System Call - Sample Program

```
starting main process
starting child process
Wed May 15 15:26:25 CDT 1985
starting parent process
parent
parent
parent
parent
parent
ending parent process
$
```

Figure 8-5. Output from forktst3 Sample Program

The child portion of this program does not contain an **exit** call because when it performs the **execl** call, the new program, **date** loads on top of the program in the child process (destroying any program code that is there). When the new program ends, it stops the child process and returns a completion code to the waiting parent process.

8.4.2.12 Example of Additional Process Handling System Calls

The program in Figure 8-6 uses the **rfork(), rexec(), run()**, and **migrate()** system calls. You can enter the program as **forktst4.c**. Compile it using the **cc** command:

```
cc -o forktst4 forkts4t.c
```

The resulting command, **forktst4**, can take an optional argument to specify the target cluster site.

To run the program for this example, enter the command:

```
forktst4 site_number
```

When you run the program shown in Figure 8-6, it displays output that looks like Figure 8-7. The sample output shown resulted from entering the command **forktst4 30** from a shell running on site number 29.

```
#include <stdio.h>
#include <sys/types.h>
#define PR(value) {fprintf(stdout, "%s\n", (value)); fflush(stdout);}

main(argc, argv)
int argc; char *argv[ ];
{
siteno_t siteno;
int status;

  if (argc > 1)

       siteno = (siteno_t) atoi(argv[1]);

  else
       siteno = (siteno_t) 0;

  PR("starting parent process");
  if (fork() == 0)
       {

       PR("starting child process");
       if (fork() == 0) {
              PR("starting grandchild process");
              migrate(siteno);
              execl("/usr/bin/sitenum", "sitenum", (char *)0);
       }
       wait(&status);
       PR("resuming child process");
       if (fork() == 0)
       {
              PR("starting grandchild process");
              rexecl("/usr/bin/sitenum", "sitenum",(char *)0,siteno);
       }
       wait(&status);
       PR("resuming child process");
       if (rfork(siteno) == 0)
       {
              PR("starting grandchild process");
```

```
                execl("/usr/bin/sitenum", "sitenum", (char *)0);
        }
        wait(&status);
        PR("resuming child process");
        PR("starting grandchild process");
        runl("/usr/bin/sitenum", "sitenum", (char *)0, siteno, 0,(char*)0);
        wait(&status);
        PR("resuming child process");
        execl("/usr/bin/sitenum", "sitenum", (char *)0);
    }
    wait(&status);
    PR("resuming parent process");
    execl("/usr/bin/sitenum", "sitenum", (char *)0);
  }
```

Figure 8-6. Additional Process Handling System Calls - Sample Program

```
  starting parent process
  starting child process
  starting grandchild process
  30
  resuming child process
  starting grandchild process
  30
  resuming child process
  starting grandchild process
  30
  resuming child process
  starting grandchild process
  30
  resuming child process
  29
  resuming parent process
  29
```

Figure 8-7. Additional Process Handling System Calls - Sample Output

8.4.2.13 Example of a Pipe System Call

The program in Figure 8-8 uses the **pipe** system call in a simple, but operating program. Compile the program using the **cc** command and copy it to its own file, **pipetst**, as in the previous **fork** example. This example implements functions similar to the **popen** and **pclose** subroutines which are described in the *AIX Operating System Technical Reference*.

```
/* pipetst.c -- sample program using the pipe system call.  The
 * routines openp and closep, defined below, are similar (but not
 * identical) in function to the popen and pclose library routines
 * provided by AIX.
 */
#include <sys/types.h>           /* for pid_t */
#include <stdio.h>               /* for printf() function */
#include <string.h>              /* for strlen() function */
#include <fcntl.h>               /* for constants O_RDONLY, O_WRONLY */

pid_t childpid, fork(), wait();

main()
{
        int w_fdesc;
        char *message = "Writing piped message.";
        int mesg_len;

        mesg_len = strlen(message);

        w_fdesc = openp("pr", O_WRONLY);
        if (w_fdesc == -1) {
               printf("error creating pipe");
               exit(1);
        }
        if (write(w_fdesc, message, mesg_len) !=  mesg_len)  {
                printf("error writing to pipe");
                exit(1);
        }

        closep(w_fdesc);
        exit (0);
}

int
openp(cmd, mode)
char *cmd;
int mode;
{
       int p_fdesc[2];

       if (pipe(p_fdesc) < 0)
               return -1;

       /* Create child process to be on the other end of the pipe */
       if ( (childpid = fork()) == 0 ) {
               if (mode == O_RDONLY) {
                   /* If caller is going to be the reader, the child
                    * is the writer so close the read end of the pipe,
                    * and move the write end of the pipe to stdout
```

```
                        * (file descriptor 1) by calling close, dup, close */
                       close(p_fdesc[O_RDONLY]);
                       close(1);
                       dup(p_fdesc[O_WRONLY]);
                       close(p_fdesc[O_WRONLY]);
                } else {
                       /* If caller is going to be the writer, the child
                        * is the reader so close the write end of the pipe,
                        * and move the read end of the pipe to stdin
                        * (file descriptor 0) by calling close, dup, close */
                      close(p_fdesc[O_WRONLY]);
                      close(0);
                      dup(p_fdesc[O_RDONLY]);
                      close(p_fdesc[O_RDONLY]);
                }
                execl("/bin/sh", "sh", "-c", cmd, 0);
                _exit(1);
        }
        if (childpid == -1)
                return -1;

        if (mode == O_RDONLY) {
                close(p_fdesc[O_WRONLY]);
                return p_fdesc[O_RDONLY];
        } else {
                close (p_fdesc[O_RDONLY]);
                return p_fdesc[O_WRONLY];
        }
  }

  int
  closep(fd)
  int fd;
  {
        pid_t retpid;
        int status;

        close(fd);
        while ((retpid = wait(&status)) != childpid) {
                if (retpid == -1)
                        return -1;
        }
        return status;
  }
```

Figure 8-8. Using the pipe System Call

To run the program, enter the command:

```
pipetst
```

This program shows how to handle pipe file descriptors to use pipe for interprocess communication. It creates a pipe, forks another process, and writes a simple message to the pipe. The child process writes the message to the screen after receiving the message through the pipe.

When the program runs, it calls **openp** to open a pipe and fork to create a

new process. The **pipe** call returns two file descriptors to the calling program: the first describes the end of the pipe to use for writing; the second describes the end of the pipe to use for reading. Both processes must cooperate to use the pipe. If one process writes to the pipe, the other process should not write to the same pipe at the same time. Similarly, if one process reads from the pipe, the other process should write something into the pipe to be read. Therefore, after the fork in this program, the child process closes the write file descriptor for the pipe and the parent process closes the read file descriptor for the pipe. The parent can then write into the pipe and the child can read from the pipe, using the remaining open file descriptors.

Note: To have two-way data transfer between processes, open two different pipes: one for reading and one for writing. Do not use one pipe for two-way data transfer.

After the child process closes the read end of the pipe, it also closes the file descriptor for standard input that belongs to that process (file descriptor 0). Then it uses a **dup** system call to copy the read file descriptor for the pipe. Because the system assigns file descriptors starting at the lowest number available and file descriptors 0 and 1 are now available, the copy of the read end of the pipe becomes standard input for the child process. The child then closes the original read file descriptor for the pipe, leaving only the file descriptor for standard input assigned to the pipe in the child process.

Having set up the pipe as standard input, the child process loads and executes (**execl**) the shell and the command that is passed to it from the parent process (**pr**). The new program in the child process then waits for input from its standard input, the read end of the pipe.

The parent process closes the read file descriptor for the pipe and returns the write file descriptor to the calling process (**pipetst**). **Pipetst** writes a message into the pipe, which is written out by the **pr** command running in the child process.

Because the **pr** command writes a formatted page output to the screen (standard output), the output may flash on the screen too fast to see it happen. If this occurs, use the command:

```
stty page length 24
```

to set the terminal to paging mode. The output fills the screen and then waits for you to press **Enter** before it displays another screen of data.

8.4.3 Process Identification

Each process in the system has a unique number (***process ID***) that the system uses to control the activities in the system. In addition to the process ID, the system also assigns the following identifiers to a process (see Figure 8-9):

Effective Group ID
The group ID that determines what files the program can access. In most cases this is the same as the real group ID; however, you can create a process that has access permission that is different from your own, and the effective group ID would be different. Refer to the **exec** system call in *AIX Operating System Technical Reference* for information about setting the set group ID bit.

Effective User ID
The user ID that determines what files the program can access. In most cases this is the same as the real user ID; however, you can create a process that has access permission that is different from your own, and the effective user ID would be different. Refer to the **exec** system call in *AIX Operating System Technical Reference* for information about setting the set user ID bit.

Parent Process ID
The process ID of the parent process that issued a **fork** call to create the process.

Process Group ID
A process ID value assigned to a collection of related processes so that they may be signalled as a group. For additional information, refer to the discussion of **setpgid** and **kill** in the *AIX Operating System Technical Reference*.

Real Group ID
The group ID of the user who started the process.

Real User ID
The user ID of the user who started the process.

Saved-Set Group ID
The value of a process' effective group ID saved when the process starts. This affects a process' ability to send and receive signals. For additional information, refer to the discussion of **kill** in the *AIX Operating System Technical Reference*.

Saved-Set User ID
The value of a process' effective user ID saved when the process starts. This affects a process' ability to send and receive signals. For additional information, refer to the discussion of **kill** in the *AIX Operating System Technical Reference*.

Session ID
A process ID that identifies all processes that started from a particular terminal.

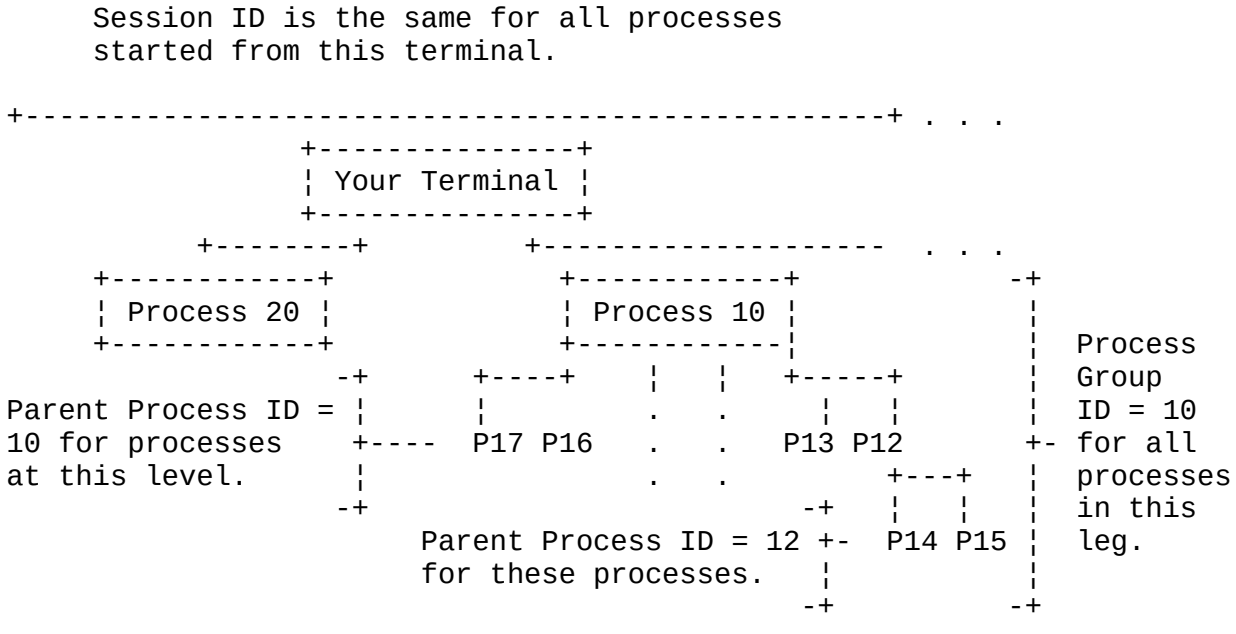
Terminal Process Group ID
The process ID of the foreground process group for a particular

terminal. The foreground process group contains all processes that can read from the terminal.

A program can determine or change process identifiers with the following system calls:

Call	Description
getegid	Gets effective group ID of the calling process.
geteuid	Gets effective user ID of the calling process.
getgid	Gets real group ID of the calling process.
getpgrp	Gets process group ID.
getpid	Gets process ID.
getppid	Gets parent process ID.
getsid	Gets session ID.
getuid	Gets real and effective, user and group IDs.
setgid	Sets group ID.
setpgrp	Sets process group ID.
setpgid	Sets process group ID of another process.
setsid	Sets session ID.
setuid	Sets user ID.
tcgetpgrp	Gets terminal process group ID.
tcsetpgrp	Sets terminal process group ID.
ulimit	Gets and sets user limits.
uname	Gets name of current system.
unamex	Gets information about current system.
usrinfo	Gets and sets user information about the owner of a process.

See *AIX Operating System Technical Reference* for complete information about any of these system calls.



Real User ID = Your ID number for all processes.
Effective User ID = Your ID number or an ID number that gives special access permissions to the process.
Real Group ID = Your group ID number for all processes.
Effective Group ID = Your group ID number or an ID number that gives special access permissions to the process.

Figure 8-9. Relationship of IDs in the System

Subtopics
8.4.3.1 Concurrent Groups

8.4.3.1 Concurrent Groups

The operating system allows you to belong to many different groups. Being a member of a group, you can access the files whose permission bits allow access to members of that group. If you belong to more than one group, you can access the files that are available to all of those groups at any time. The ability to access files from many groups at the same time is called ***concurrent groups***.

When you log in to the system, the **login** program checks **/etc/passwd** and **/etc/group** to determine group membership. The **login** program then sets up access to the files of those groups, and assigns a group ID to indicate which of the groups is the ***primary group***. When you create files, those files become available to other members of the primary group. To change the primary group, use the **newgrp** command to change the group ID to that of another group to which you belong.

You can run a program only if you own it and have user execute permission, if it is available to one of your groups, or if the *others* permission of the program allows you to execute it. When the program runs, it behaves much like a user with respect to groups. It normally takes on the group memberships of the person running the program. For example, if you run a program, that program can access all files that you can access; if another user runs the same program, the program no longer can access your files, but can access the files that the new user can access. You can change this behavior by using the **chmod** command to set the program to always run under its own set of user and group IDs. See *AIX Operating System Commands Reference* for information about the **chmod** command.

Programs can determine or change a process' concurrent groups using **getgroups** and **setgroups**. Refer to the *AIX Operating System Technical Reference*.

8.4.4 Process Attribute Calls

The following describes additional attributes in the environment.

Subtopics

8.4.4.1 <LOCAL> Alias

This attribute is evaluated whenever the system encounters a symbolic link beginning with <LOCAL>. The alias is then substituted for the token "<LOCAL>" during path name resolution. This allows the same symbolic link to refer to different files for processes running on different sites in a TCF cluster.

8.4.4.2 Cluster Site Number

This number uniquely identifies the cluster site within the cluster.

8.4.4.3 Cluster Site Path List

The site path list is used to resolve file names that refer to hidden directories. When the system encounters a hidden directory while resolving a path name, it extracts elements of the site path until it finds a match between a list element and a file in the hidden directory. If the system finds such a match, it selects that file and finishes resolving the path name. The site path list is especially useful for selecting executables in an environment of heterogeneous machine types.

8.4.4.4 Cluster Site Permission Mask

The site permission mask identifies the cluster sites to which a process can move or create new processes.

The following calls deal with process attributes:

Call	Description
getlocal	Gets the <LOCAL> alias.
getspath	Gets the site path list.
getxperm	Gets the site permission mask.
setlocal	Sets the <LOCAL> alias.
setspath	Sets the site path list.
setxperm	Sets the site permission mask.
site	Gets the site number of the specified process.

8.4.4.5 Example of Process ID and Attribute Calls

The code in Figure 8-10 uses some of the process ID and attribute system calls in a simple program. You can enter the program as **ppidtst.c**. Compile the program using the **cc** command:

```
cc -o ppidtst ppidtst.c
```

To run the program, enter the command:

```
ppidtst site_number
```

Figure 8-11 shows the output for the program when run as:

```
ppidtst 10
```

from a C-shell on site number 3.

When the program runs, it writes output to the display. The program creates two processes: a child and a grandchild. Each process writes messages to the display to indicate the following:

Child process ID (except for the grandchild process

Process I

Process group I

Parent process I

Site number on which the process is running

The processes cooperate so that the output to the display is orderly. The **ps** (process status) command reports the process id of active processes. Use the **ps** command to verify that the shell is the parent process of the parent process when you run the example.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#define PR(value) fprintf(stdout, "%s\n", (value))
#define PRT(s1, v1) fprintf(stdout, "%s %d.\n", (s1), (v1))

main(argc, argv)
int argc; char *argv[ ];
{
  int status1;
  int status2;
  pid_t frkpid1;
  pid_t frkpid2;
  pid_t chldpd;
  pid_t g_chldpd;
  siteno_t siteno;
     if (argc > 1) {

          siteno = (siteno_t) atoi(argv[1]);
```

```
        }
        else
              siteno = (siteno_t) 0;

        PR("starting main process");
        fflush (stdout);
        if ((frkpid1 = fork()) == 0) {
           if ((frkpid2 = rfork(siteno)) == 0) {

                PRT("grandchild: my process id is ", getpid());
                PRT("grandchild: my parent process id is ", getppid());
                PRT("grandchild: my process group id is ", getpgrp());
                PRT("grandchild: my site is ", site(getpid()));
                 PR("ending grandchild process");
                exit(0);
             }
             g_chldpd = wait(&status2);
             PRT("child: my child's process id is ", g_chldpd);
             PRT("child: my process id is ", getpid());
             PRT("child: my parent process id is ", getppid());
             PRT("child: my process group id is ", getpgrp());
             PRT("child: my site is ", site(getpid()));
             PR("ending child process");
             exit(0);
        }
        chldpd = wait(&status1);
        PRT("parent: my child's process id is ", chldpd);
        PRT("parent: my process id is ", getpid());
        PRT("parent: my parent process id is ", getppid());
        PRT("parent: my process group id is ", getpgrp());
        PRT("parent: my site is ", site(getpid()));
        PR("ending parent process");
  }
```

Figure 8-10. Process ID and Attribute Calls Sample Program

```
  starting main process
  grandchild: my process id is 321134
  grandchild: my parent process id is 321133
  grandchild: my process group id is 321132
  grandchild: my site is 10
  ending grandchild process
  child: my child's process id is 321134
  child: my process id is 321133
  child: my parent process id is 321132
  child: my process group id is 321132
  child: my site is 3
  ending child process
  parent: my child's process id is 321133
  parent: my process id is 321132
  parent: my parent's process id is 320321
  parent: my process group id is 321132
  parent: my site is 3
  ending parent process
```

Figure 8-11. Process ID and Attribute Calls Sample Program

8.4.4.6 Changing the Controlling Terminal

Typically, each process is associated with a process group and has a controlling terminal. Each process group has a group leader process. For example, when you log in, the shell process is the process group leader and any process descendants are in that process group. The terminal you are typing on is the controlling terminal.

However, it is sometimes necessary for a program to establish the controlling terminal on its own or to disassociate itself from the process group and not have a controlling terminal. You use the **setpgrp** system call in both cases, and in both cases it is important that you perform a series of steps completely and in the correct order. If you do not follow the correct procedures, problems can occur that are sometimes intermittent and always very difficult to diagnose.

Establishing a Controlling Terminal: To establish a controlling terminal, perform the following steps in the following order:

1. Close all the file descriptors of the controlling terminal for the current process, if there are any.

2. Issue the **setpgrp** system call. This makes the current process the group leader.

3. Open the desired terminal. If this terminal is not already a controlling terminal for some other process group, it becomes the controlling terminal for this process group. The rule is as follows: The first group leader process to open a terminal that is not already a controlling terminal, acquires that terminal as a controlling terminal for that process group.

4. Issue the **dup** system call so that additional file descriptors also refer to that terminal. File descriptors 0, 1, and 2 refer to the terminal as the default.

The following program fragment illustrates these steps:

```
#include <fcntl.h>

close(0);
close(1);
close(2);

setpgrp();

if ( open("/dev/tty0",O_RDWR) == -1)
  return(errno);
else
{
  dup(0);
  dup(0);
}
```

If you fail to establish the process group correctly, some functions behave improperly. For example, the **SIGINT** signal is sent to all processes in the process group and if the process group is incorrect for a particular terminal, the signal is not sent or is sent to the wrong group

of processes. The **SIGHUP** signal is sent to all processes in the process group when the group leader exits and if the process group is incorrect, the signal behaves incorrectly.

Eliminating the Controlling Terminal: It is sometimes necessary that a program not have a process group and a controlling terminal. For example, a program that runs in the background might want to write error or information messages to a terminal (such as the system console) but not accept information from the terminal.

To eliminate the controlling terminal, perform the following steps in the following order:

1. Close all the file descriptors for the controlling terminal for the current process if there are any. Do this even if the inherited files are for the same terminal as the desired output terminal.

2. Open the desired output terminal.

3. Issue the **setpgrp** system call. This makes the current process the group leader. This also means that the terminal opened in the previous step is not the controlling terminal. In fact, this process group has no controlling terminal.

The following program fragment illustrates these steps:

```
#include <fcntl.h>

close(0);
close(1);
close(2);

if ( open("/dev/console",O_RDWR) == -1)
  return(errno);

setpgrp();
dup(0);
dup(0);
```

In this situation, the program can write to the terminal (system console) but should not read from it. The interrupt key from the terminal does not affect the program because the terminal is not the controlling terminal for that process group. It is not normally possible for another process to obtain the system console as a controlling terminal.

A typical scenario would be where you invoke a background program from the /etc/rc shell and the background program, traditionally called a ***daemon***, uses **/dev/console** as an output device for messages. **/dev/console** is enabled to allow another user to log into it.

Failure to observe the above rules could cause the following undesirable behavior for the user who managed to log into terminal **/dev/console**:

The interrupt key might not work as expected because the terminal i not the controlling terminal for the proper group.

Many commands would fail because **/dev/tty** could not be opened. **/dev/tty** is a pseudo device that is the controlling terminal for some

command processes. If the process has no controlling terminal, the open for **/dev/tty** fails.

8.4.5 Process Tracking Calls

The system provides the following calls to monitor the operation of a process or group of processes in the system. These calls are the means by which the system commands of similar names are created. Use these calls within a program to gain information about how the program runs in each section of the program. They are primarily tools for use during development which would not be included in the final version of the program. See *AIX Operating System Technical Reference* for information about the syntax and flags of these calls.

Call	Description
acct	Enables/disables process accounting.
getrusage	Gets information about resource utilization.
profil	Provides an execution time profile.
ptrace	Provides a process trace.
times	Gets process and child process times.

8.5 Interprocess Communications

In addition to pipes previously described in "Example of a Pipe System Call" in topic 8.4.2.13, the system provides many methods for two or more processes to communicate with each other. It provides system calls for:

Sending and receiving signal

Setting and reading semaphore

Sending and receiving messages via UNIX System V message queue

Sending packets via TCP/IP socket communication

Each of these sets of calls provides one method of sending small pieces of information between processes. For larger blocks of data, use the shared memory facility described in "Shared Memory Calls" in topic 8.5.6.

Subtopics

8.5.1 Signal Calls

Signals provide a simple method of communication between two processes. Using signals, one process can inform another process of status conditions that occur during the process, or can tell the other process when an event occurs. Use signals to activate a process for error recovery, or to help control access to shared resources (see "Semaphore Calls" in topic 8.5.3 and "Shared Memory Calls" in topic 8.5.6 for more advanced control of shared resources). The signals that a program can use are defined in the system file, **/usr/include/signal.h** (see **sigaction** in *AIX Operating System Technical Reference*). The system calls that allow a program to use signals are:

Call	**Description**
alarm	Sets a process alarm clock.
kill	Sends signal to process(es).
kill, kill3	Sends signal to process or process group. The variant **kill3** (**pid, sig, arg**) allows an additional argument to be passed to a signal handler.
pause	Suspends process until signal.

Call	**Description**
signal, sigvec, sigaction	Specifies what to do when the process receives a signal.
sigpending	Examines pending signals.
sigprocmask, sigsetmask, sigblock	Sets the current signal mask.
sigstack	Sets and gets signal stack context.
sigsuspend, sigpause	Automatically changes the set of blocked signals and waits for an interrupt.
sigemptyset, sigfillset, sigaddset, sigdelset, sigismember	Creates and manipulates signal masks.

The signal is an integer value. The meanings of these values are defined in the header file **/usr/include/signal.h**. The system generates most of these signals to tell a program of error or status conditions in the system. A program should be able to respond to these conditions. The program can, however, use a few of the signals for its own purposes. Figure 8-12 lists the user signals.

Figure 8-12. User Controlled Signals

Signal	**Definition**
SIGALRM	Use the **alarm** system call to set a time value. The system sends SIGALRM to the program when that time is up.
SIGUSR1	A signal defined by cooperating processes to mean the same thing to those processes. One process can generate this

signal using the **kill** system call. The other cooperating processes that receive the signal can then react to the condition indicated by that signal.

SIGUSR2 Another signal defined by cooperating processes.

Subtopics
8.5.1.1 How to React to a Signal
8.5.1.2 Example of Trapping a Signal

8.5.1.1 How to React to a Signal

If a program does not have code to handle signals that it could receive, it may end when it receives a signal. Therefore, include enough code to recognize all signals that the program might receive (including system error signals). Use the **signal** system call to define the actions to be performed when the program receives a particular signal. The format of the **signal** call is:

```
signal(sig,func);
```

where:

sig Is the integer signal number or the system macro name that represents that value (defined in **/usr/include/signal.h**).

func Is a function code to indicate what action to take when you receive the signal.

The following are the codes that can be received and the appropriate actions for each code:

Code	Action
SIG_DFL	Performs the default action when the signal is received. This is generally process termination. For some signals, the default action is one of the following: Ignore the signal Stop the process Terminate the process with a core dump Migrate the process to another cluster site.
SIG_IGN	Ignores the indicated signal.
Function Address	Executes the signal-catching function pointed to by **func** when the signal is received.

For example,

```
signal(SIGPIPE,SIG_IGN);
```

tells your program to ignore a signal that indicates a pipe error.

8.5.1.2 Example of Trapping a Signal

The program in Figure 8-13 uses the signal system call in a simple program that runs. Compile the program using the **cc** command with the output going to the file, **sigtst**, as in the previous **fork** example.

To run the program, enter the command:

```
sigtst
```

When the program runs, it asks you to press any letter key (the **e** letter key to exit the program). Whatever letter you enter, the program echoes to the screen. That is the main part of the program, but it does not use signal processing.

You can generate an interrupt signal to the program (SIGINT) by pressing the key that the driver program for the terminal defines as sending that signal. This key is **DEL** on System V, **Ctrl-C** on 4.3BSD, and the **Ctrl-Bksp** key sequence on HFT-supported systems. When the program receives this signal, it branches to a routine that writes a message to the screen. The program then returns to its main program operation.

Although this function is not very useful, it shows how to build interrupt handling routines into a program. The program first tests whether interrupts (SIGINT) are ignored, and creates the jump to the interrupt trap routine only if interrupts are not ignored.

```
   /* assign the previous value of */
   /* SIGINT to istat/*
  istat = signal(SIGINT, SIG_IGN);
   .
   .
   .
  /* check previous condition of SIGINT */
  if(istat != SIG_IGN)
          signal(SIGINT,onintr);
          /* jump to onintr when an  */
          /* interrupt signal occurs */
```

The program makes this check because, if the program is run in the background from the Bourne shell, the shell sets SIGINT to be ignored. Therefore, the system only passes the interrupt signal to foreground processes. Testing this signal condition ensures the program will not get an interrupt signal when running in background. Note that the shell sets SIGQUIT to be ignored for background processes as well, so that the same signal testing algorithm would apply to SIGQUIT. The C shell has job control and signals are not delivered to background processes.

The program also uses the **setjmp** and **longjmp** library functions to handle the return to the main program. Before the program begins running the main loop, it stores the machine status in a structure **sjbuf**. The structure type, **jmp_buf** is defined in the system file **setjmp.h**. When the interrupt occurs, the system transfers control to the **onintr** routine. This routine is a dummy interrupt handling routine. When it finishes, it uses the **longjmp** function to set the machine status to the conditions

stored in the **sjbuf** structure. The main routine begins running again as if an interrupt had not occurred.

Each time the program returns from the **onintr** routine, it sets up the interrupt signal condition again. If it did not do this, it would not receive any more interrupt signals. The system sends only one interrupt signal for each request it receives. Once it sends that signal, the program must request another. If another interrupt signal occurs before the program sets up to catch the interrupt signal, the program stops. The enhanced signal facility (see "Enhanced Signal Facility" in topic 8.5.2) provides a way to avoid this problem.

```
#include <sys/signal.h>
#include <setjmp.h>
#include <stdio.h>
#include <errno.h>

jmp_buf sjbuf;

void
onintr(signum)
{
    printf("SIGNAL %d received\n", signum);
    longjmp(sjbuf, 1);
}

main()
{
    void (*istat) ();
    int c = 0;

    /* Assign previous value of SIGINT to istat */
    istat = signal(SIGINT, SIG_IGN);

    /* Return here after processing interrupt */
    (void) setjmp(sjbuf);

    /* check previous condition of SIGINT */
    if (istat != SIG_IGN)
        signal(SIGINT, onintr);
        /* jump to onintr when an interrupt signal occurs */

    printf("Please enter a letter (e to exit)\n");

    while (c != 'e') {
        c = fgetc(stdin);
        if (c == -1) {
            fprintf(stderr, "main: fgetc() encountered an error %d\n", errno
            exit(1);
        }
        if (c != '\n' && c != 'e') {
            printf("You typed the following letter: %c\n", c);
            printf("Please enter a letter (e to exit)\n");
        }
    }
}
```

Figure 8-13. Example of Signal Trapping

8.5.2 Enhanced Signal Facility

In addition to the standard signal facilities described in "Signal Calls" in topic 8.5.1, the system provides an enhanced signal facility. The enhanced facility treats signals received from the system in a way similar to the handling of hardware interrupts, allowing a program to mask each type of signal while it is processing. The facility allows for up to 64 different signals, but only those defined in the file **/usr/include/signal.h** can be used. These are the same signals that the standard signal facility uses. You can choose to use either facility in a program, but the enhanced signal facility is recommended for new applications, with the standard signal facility being provided as a compatibility interface for old programs.

The system calls that comprise the enhanced signal facility are listed in Figure 8-14. *AIX Operating System Technical Reference* provides complete information for each of these calls.

Figure 8-14. Enhanced Signal Calls

Call	**Function**
sigprocmask	Sets the signal mask to a new value.
sigsuspend	Sets the signal mask to a new value, waits for a signal allowed by the new mask and restores the old mask value when a signal is received.
sigaction	Establishes conditions to handle a specified signal.
execve	Starts a new program in the current process, resets all signals that are being caught by the old program to terminate the new program, resets signal stack state, and retains the current signal mask value.
sigemptyset	Initializes a signal mask variable so that all signals are excluded (unmasked).
sigfillset	Initializes a signal mask variable so that all signals are included (masked).
sigaddset	Adds a signal to a signal mask variable.
sigdelset	Deletes a signal from a signal mask variable.
sigismember	Tests whether a specific signal is masked within a given signal mask variable.

Subtopics

8.5.2.1 Responding to Signals

A program can receive signals from the sources shown in Figure 8-15.

Figure 8-15. Sources of Signals

Source	**Description**
Program faults	A programming error such as an illegal instruction or memory reference produces this type of message.
Terminal keys	Special key sequences generate signals to running processes in the system that belong to the terminal group of the terminal. These signals include **INTERRUPT**, **QUIT**, **HANGUP**, **STOP**, or **KILL** (the **KILL** signal cannot be masked).
Hardware exceptions	Hardware faults generate signals to running processes.
System	The system can generate signals, such as the death of a child process, to aid in process control.
Other processes	Other processes can send signals to a process to coordinate system activities.

The description of **func** in "How to React to a Signal" in topic 8.5.1.1 contains the defined responses for responding to a signal once a program receives it. Each of these responses is defined in the header file **signal.h** as a constant-valued pointer that indicates a special action for the signal. If a pointer to a function is supplied instead of one of these constant pointers, that function will be invoked when the signal is received. For example, **signal.h** defines SIG_DFL as follows:

```
#define  SIG_DFL     ((VOID (*) () ) 0)
```

Defining the responses as pointers to functions allows you to use them in the **sigaction** structure (see "Using Enhanced Signals" in topic 8.5.2.2) as the pointer to the handler routine. See "Example Programs" in topic 8.5.2.6 for a sample of using the responses in the structure.

8.5.2.2 Using Enhanced Signals

When a process receives a signal, the system automatically:

1. Blocks another signal of that type from being sent to the process.

2. Saves the environment of the process.

3. Builds a new environment for the process to respond to the signal.

4. Transfers control to the signal ***handler*** routine in the process.

The signal ***handler*** is a routine that you provide to respond to the receipt of a signal. It may be a complex error recovery routine, or it may be SIG_DFL or SIG_IGN. You choose how the program responds to each of the signals. Once you have created the handler routines for all of the signals that you expect to receive, tell the system how to handle each of the signals. First, include the header file **signal.h** with the following statement:

```
#include <signal.h>
```

This file contains definitions for all the constant names used by the signal handling facility. In addition, this file contains the following structure definition:

```
struct  sigaction
{
        VOID      (*sa_handler) ( );
        int       sa_flags;
        sigset_t  sa_mask;
};
```

This is the structure type that passes information to the system when using the **sigaction** system call. It contains three members as shown in Figure 8-16.

Figure 8-16. sigvec Structure Members

Name	Description
sa_handler	A pointer to the routine that you have created to handle the signal processing for a particular signal, or one of **SIG_DFL** or **SIG_IGN** defined in **signal.h**.
sa_flags	A set of flags which determine special options to how the signals should work (primarily provided so that compatibility modes may be implemented). The possible flag bits which may be OR'd together and set in sa_flags include **SA_RESTART**, **SA_ONSTACK**, **SA_OLDSTYLE**, and **SA_NOCLDSTOP** which are defined in **signal.h**.
sa_mask	A signal mask variable which specifies which signals (in addition to those in the process' signal mask) should be blocked from being delivered when the signal handler function specified in sa_handler is executing. The sa_mask field should be set, cleared and tested using the **sigemptyset**, **sigfillset**, **sigaddset**, **sigdelset**, and **sigismember** functions.

For example, to set up a routine to receive the SIGALRM signal, use the following system call:

```
sigaction ( SIGALRM , new_act , old_act );
```

The parameters for this call have the following meaning:

Parameter	Meaning
SIGALRM	The name of the signal, SIGALRM, which is the alarm clock.
new_act	A pointer to the structure of type **sigaction** that defines the information for the signal handler.
old_act	A pointer to a structure of type **sigaction** where the system returns the values it is using for the specified signal. If this pointer is **NULL**, the system ignores it.

8.5.2.3 Waiting for a Signal

The **sigsuspend** system call stops processing in a program to wait for the occurrence of any or all of the signals, and then resume processing without altering the signal mask (**sv_mask**) with which the program normally operates. The following sequence illustrates how to stop program operation, wait for any signal to occur, and then resume program operation:

1. Program begins.

2. Program issues **sigsuspend** call with a mask value of zero to enable catching all signals:

```
sigset_t nomask;

sigemptyset (&nomask);
sigsuspend (&nomask);
```

3. A signal occurs for the process.

4. The **sigsuspend** call returns with a **-1** return code, and **errno** set to EINTR.

5. The system restores the previous signal mask value.

6. The program continues.

8.5.2.4 Protecting Important Program Events

During some parts of a program, you may want to continue processing in spite of any signals that the program may receive. Some activities, such as processing a linked list, could be difficult or impossible to recover from if the program were interrupted at that point. To protect sections of the program from interruption from all but the most serious signals (SIGKILL cannot be masked), use the **sigprocmask** system call as shown in the following sequence:

1. At the beginning of the important section of the program, use the **sigblock** system call with a mask value that, when ORed with the current mask, blocks all expected signals. Provide a variable of type **sigset_t** to hold the returned value (the old mask value) for later use:

   ```
   sigprocmask (SIG_BLOCK, &block_mask, &old_mask);
   ```

2. Process the important section of the program.

3. Restore the program to normal operation with the **sigprocmask** system call with the **SIG_SETMASK** option this time. Restore the signal mask value from the mask saved in step 1.

   ```
   sigprocmask (SIG_SETMASK, &old_mask, NULL);
   ```

4. Continue normal program operation.

8.5.2.5 Finding Out the Current Signal Mask

Each process running in the system has its own signal mask, regardless of whether or not the process uses signals. If the program does not use the signal calls to change this mask, the system assigns a value of zero to the signal mask (no signals blocked).

Two system calls provide information about the signal conditions that a program is using for a specified signal. The first call, **sigaction**, requires a defined structure of type **sigaction** where the system can write the current information. The following example tells the system to store the current information for the signal SIGINT in the structure **old_act**. Because the second parameter that normally points to the new signal structure is NULL, the current signal structure for SIGINT does not change, but it is copied to the structure **old_act**.

```
sigaction ( SIGINT , NULL , &old_act );
```

You can use the **sigprocmask** system call to get only the value of the current mask:

```
sigprocmask (SIG_BLOCK, NULL, &old_mask);
```

As with **sigaction**, since the second parameter which usually points to the new mask is NULL, the current process mask is unchanged. However, it is copied into old_mask.

8.5.2.6 Example Programs

Figure 8-17 shows an example program using enhanced signals. This program performs the same function as the example program used for signals in "Example of Trapping a Signal" in topic 8.5.1.2. Refer to the program description in that section. This program is stored as **newsig.c** in the example program directory.

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
int c;
int i;
jmp_buf sjbuf;

main()
{
   void onintr();
   int fgetc();
   struct sigaction oldact, newact;
   newact.sa_handler = SIG_IGN;
   newact.sa_flags = 0;
   sigemptyset (&newact.sa_mask);
   sigaction ( SIGINT, NULL, &oldact );
   if( oldact.sa_handler != SIG_IGN )
   {
      newact.sa_handler = onintr;
      sigaction ( SIGINT, &newact, NULL );
   }

   setjmp( sjbuf );
   printf( "%s\n", "Please enter a letter (e to exit)." );
   do
   {
      c = fgetc( stdin );
      if( c != '\n' && c != 'e' )
      {
        printf( "%s%c\n", "You typed the following letter: ", c );
        i++;
        printf( "%d\n", i );
        printf( "%s\n", "Please enter a letter (e to exit)." );
      }
   }  while ( c != 'e');
}          /* End Main */

void
onintr()
{
   printf( "%s\n", "Performing processing upon interrupt." );
   printf( "%s\n", "Finished interrupt processing; returning to main." );
   longjmp( sjbuf, 1 );
}                /* End onintr */
```

Figure 8-17. Enhanced Signals Example Program

8.5.3 Semaphore Calls

Semaphores provide a general method of communication between two processes that is an extension of the features of signals. Use semaphores in much the same way as signals, except that semaphores are:

More flexible: Processes can define a semaphore to mean what they want it to mean.

More controllable: Programs have direct control over semaphores and do not need to depend on the system to generate them.

Broader in scope: A semaphore can be any integer value, not just 1 or 0. Use them for counting as well as process coordination (see program example).

The system calls that allow a program to use semaphores are:

Call	Description
semctl	Semaphore control operations.
semget	Get set of semaphores.
semop	Semaphore operations.

Subtopics
8.5.3.1 Structure of a Semaphore Set
8.5.3.2 How to Use Semaphores
8.5.3.3 Example of Semaphores
8.5.3.4 Example of Semaphore Programming

8.5.3.1 Structure of a Semaphore Set

When using the **semget** system call to create a set of semaphores, the system returns an integer that is the ***semid***, or semaphore ID, for the set of semaphores that were created. Each semid points to a set of semaphores and a data structure that contains information about the semaphores. The data structure for semid is shown in Figure 8-18. See Figure 8-19 for the data structure of a semaphore.

Figure 8-18. Semid Data Structure

Name	**Function**
sem_perm	Operation Permission Struct
cuid	creator user ID
cgid	creator group ID
uid	user ID
gid	group ID
mode	read and alter permission
sem_nsems	Number of Semaphores in the Set
sem_otime	Time of the Last Operation (seconds since 1/1/1970)
sem_ctime	Time of the Last Change (seconds since 1/1/1970)

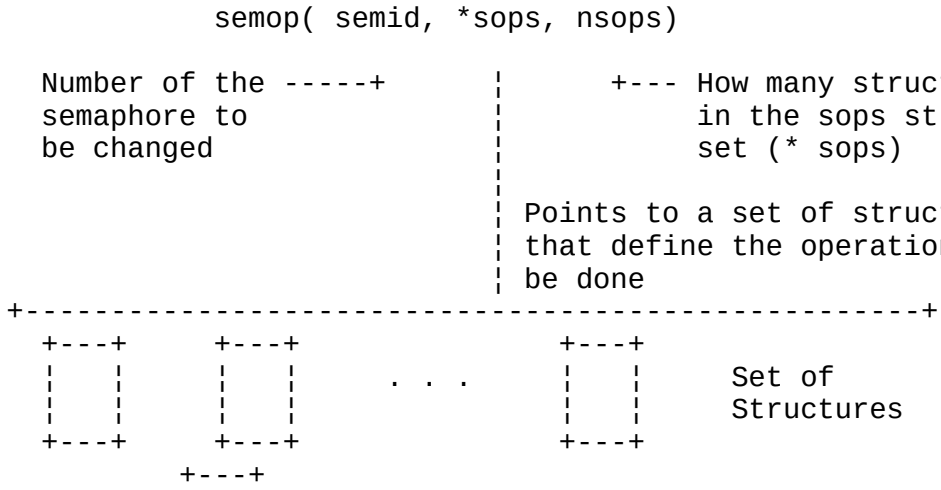
Figure 8-19. Semaphore Structure

Name	**Function**
semval	Value of the semaphore (0 or positive)
sempid	Process ID of the Last Operation
semncnt	The number of processes that are waiting for semval to be > current value of its last semop() call
semzcnt	The number of processes that are waiting for semval to be = 0.

8.5.3.2 How to Use Semaphores

Semaphores are counters that a program can test and change with a single system call (**semop**). Specify the amount of the change in the **semop** call using the **sem_op** variable within the **sops** structure as shown in Figure 8-20. When using this call, the system tests the value of **sem_op** against the value of the semaphore indicated by **sem_num**. If (sem_flg & IPC_NOWAIT) is true, the call returns without further action. If it is false, the table in Figure 8-21 summarizes the actions that occur.

Use semaphores for passing data between processes and for other one time data transfers. You can also use them to control access to a limited resource, such as a shared buffer.



One Typical Structure:

Name	**Function**
sem_num	Number of the semaphore to be changed
sem_op	The semaphore operation
sem_fig	Operation flags

Figure 8-20. Semop System Call Parameters

Figure 8-21. How Sem_op Specifies a Semaphore Operation

Value of sem_op	**Relationship to semval**	**Actions**
< 0	|sem_op| <= semval	1. semval = semval - |sem_op|
	|sem_op| > semval	1. semncnt = semncnt + 1 2. wait for semval >=|sem_op|; then

		semval = semval - |sem_op| semncnt = semncnt -1
< 0		1. semval = semval + sem_op
0	semval==0	1. return
	semval !=0	1. semzcnt = semzcnt + 1 2. wait for semval = 0; then semzcnt = semzcnt -1 return

8.5.3.3 Example of Semaphores

For example, two processes **proca** and **procb** share a buffer **bufr**. **proca** produces data packages and places each package in the buffer as it produces that package. **procb** uses the data packages from the buffer in its operation, but does not use them at the same rate that **proca** puts them in the buffer. The buffer can hold only three data packages. The processes use two semaphores to ensure that:

proca does not try to put a package in a full buffer

procb does not try to take a package from an empty buffer.

The processes agree that the semaphores have the following meanings:

sem1 The number of empty slots in the buffer. If sem1 is greater than zero, then **proca** can put a package in the buffer.

sem2 The number of data packages in the buffer. If sem2 is greater than zero, then **procb** can take a package from the buffer.

Figure 8-22 shows the relation of the processes to the buffer and semaphores. When the processes start, **sem1** has a value of 3 (empty slots) and **sem2** has a value of 0 (packages in **bufr**).

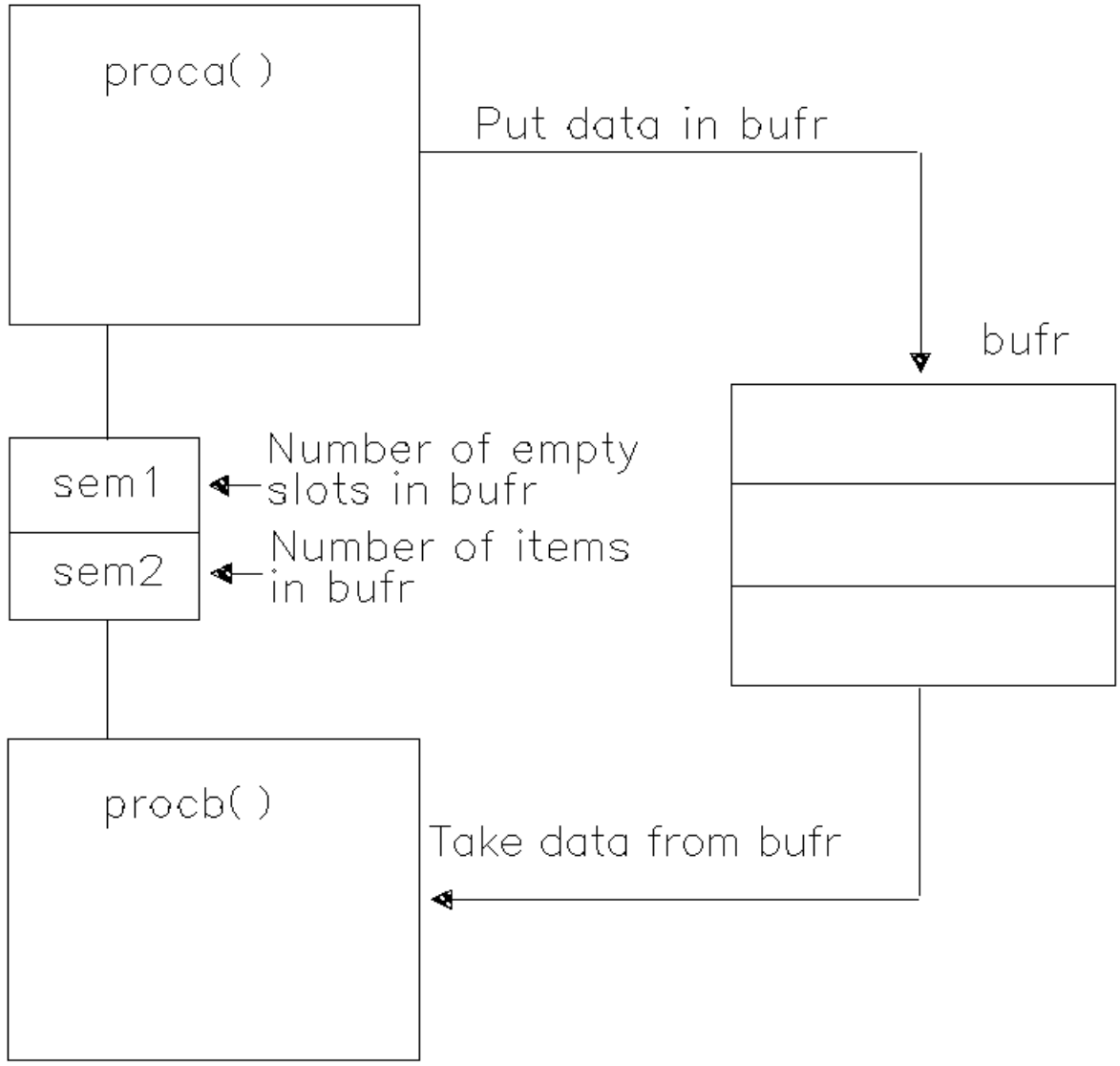


Figure 8-22. Using Semaphores Concept Example

proca: **proca** puts items into the buffer. Before it can put an item in the buffer, it tests **sem1** to find out if there is room in the buffer for the item. If **sem1** is greater than, or equal to 1 (the number of items to be put into **bufr**), then it:

1. Decrements **sem1** by 1.
2. Puts an item in the buffer.
3. Increments **sem2** by 1.

If **sem1** is less than 1, **proca** waits until there is room in the buffer (**sem1** >= 1), and then performs the preceding steps. If **proca** has to wait, it increments the **semncnt** flag for **sem1** to indicate that it is waiting. It decrements this flag when it continues. Figure 8-23 shows the **semop** call and structure that handles all of the semaphore operations for **proca**.

```
semop( xx , *sops,   2 )
```

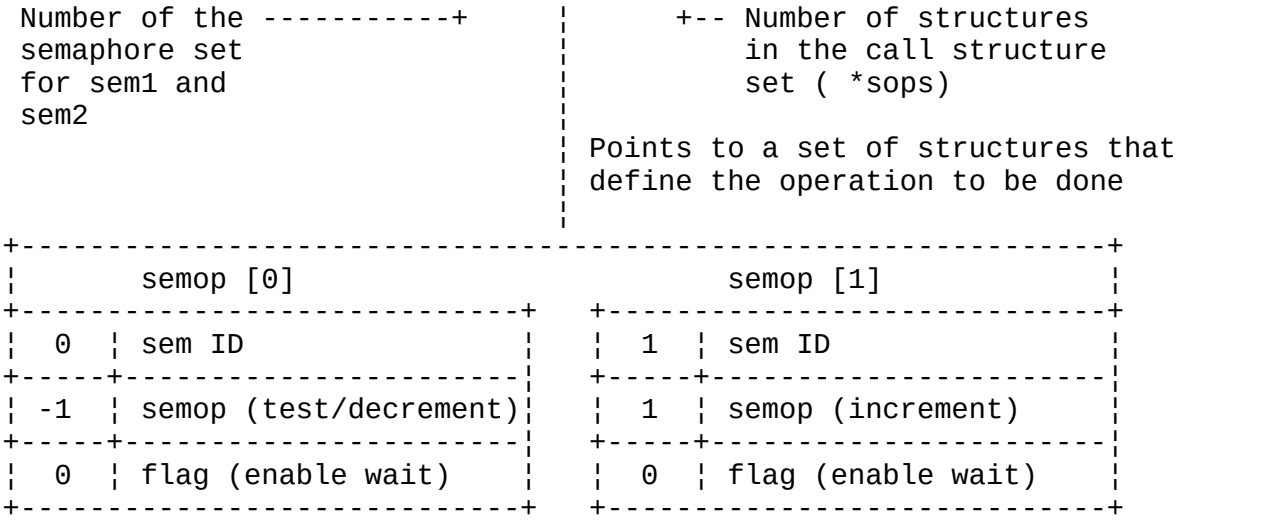


Figure 8-23. Semop Call for Proca

procb: **procb** takes items from the buffer. Before it can take an item, it tests **sem2** to find out if there is anything in the buffer. If **sem2** is greater than, or equal to 1 (the number of items to be taken), then it:

1. Decrements **sem2** by 1
2. Takes an item from the buffer.
3. Increments **sem1** by 1.

If **sem2** is less than 1, **procb** waits until there is something in the buffer (**sem2** >= 1), and then performs the preceding steps. If **procb** has to wait, it increments the **semncnt** flag for **sem2** to indicate that it is waiting. It decrements this flag when it continues. Figure 8-24 shows the **semop** call and structure that handles all of the semaphore operations for **procb**.

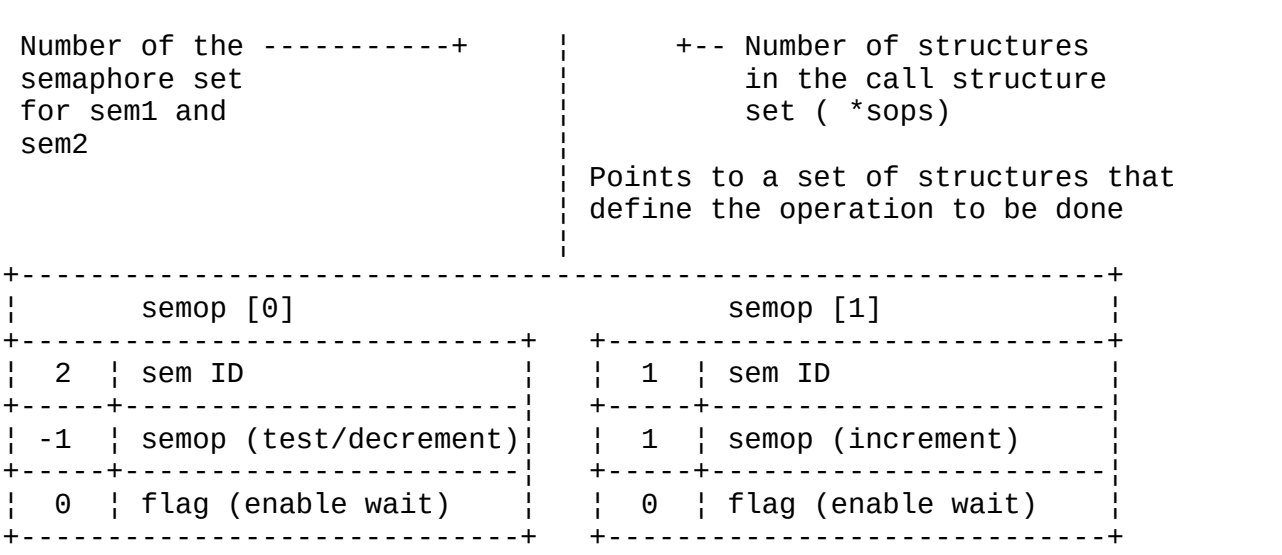


Figure 8-24. Semop Call for Procb

Operation: The chart in Figure 8-25 shows the operation of the semaphores

to control access to the buffer.

Figure 8-25. Semaphore Usage

Event	sem1		sem2	
	Value	**semncnt**	**Value**	**semncnt**
1. Start: The initial state of the parameters	3	0	0	0
2. Procb tries to get item:	3	0	0	1
3. Proca puts item in bufr.	2	0	1	1
4. Procb can now get item.	3	0	0	0
5. Proca puts item in bufr.	2	0	1	0
6. Proca puts item in bufr.	1	0	2	0
7. Proca puts item in bufr.	0	0	3	0
8. Proca tries to put item in bufr.	0	1	3	0
9. Procb gets item from bufr.	1	1	2	0
10. Proca can now put item in bufr.	0	0	3	0

8.5.3.4 Example of Semaphore Programming

The program in Figure 8-26 shows the use of semaphores in a situation similar to the example concept described previously. This program also uses some shared memory calls as described in "Shared Memory Calls" in topic 8.5.6. You can compile and run this program to see the effects of the system calls.

```
/* sem_test.c -- sample semaphore test program */

#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>


/* The following structure, STACK, will be placed in a shared memory
 * segment and access to it will be controlled by two semaphores:
 *      SN_FREE -- a count of the number of free stack frames, and
 *      SN_USED -- a count of the number of used stack frames.
 *
 * Each stack entry will be a string of up to MESSAGE_LEN characters.
 */

#define STACK_SIZE              2
#define MESSAGE_LEN             100
struct STACK {
        struct STACK_ITEM {
                char message[MESSAGE_LEN];
        } stk[STACK_SIZE];
        int sp;
} *stack;

/* Semaphore numbers */
#define SN_FREE          0
#define SN_USED          1

/* Shared Memory Address (0 means let the system pick) */
#define MADDR            0

int semid;
int shmid;

main ()
{
       char *pop_item();
       int semget();
       pid_t pid_1, pid_2, retpid, fork(), wait();
       int stts_1, stts_2, status;
       key_t key, ftok();
       char *shmat();
       int shmdt();
       int nsems;
```

```
/* Create a semaphore set with 2 semaphores */
key = ftok("sem_test", 's');
nsems = 2;
semid = semget(key, nsems, IPC_CREAT ¦ S_IRUSR ¦ S_IWUSR);
semctl(semid, SN_FREE, SETVAL, STACK_SIZE);
semctl(semid, SN_USED, SETVAL, 0);


/* Create a shared memory segment */
shmid = shmget(key, sizeof(struct STACK), IPC_CREAT¦S_IRUSR¦S_IWUSR);
stack = (struct STACK *) shmat(shmid, MADDR, 0);
stack-> sp = 0; /* stack starts off empty */


/* First child, will add messages to the stack */
if ((pid_1 = fork()) == 0) {
        int i;

        for (i = 0; i <= 5; i++) {
             p(SN_FREE);   /* reserve a spot in the stack */
             printf("Producer: Sending item number %d\n", i);
             push_item("Test Item");
             v(SN_USED);  /* announce availability of the item */
        }
        exit(0);
}

/* Second child, the consumer, will remove messages from the stack */
if ((pid_2 = fork()) == 0) {
        int i;
        char *item_ptr;

        for (i = 0; i <= 5; i++) {
               p(SN_USED);  /* get permission to take an item */
               item_ptr = pop_item();
               printf("Consumer: Got item number %d; item is '%s'\n",
                                i, item_ptr);
               v(SN_FREE);    /* announce that item is removed */
        }
        exit(0);

}


/* Wait for child processes to finish */
stts_1 = stts_2 = -1;
while ((retpid = wait(&status)) != -1) {
        if (retpid == pid_1)
                stts_1 = status;
        else if (retpid == pid_2)
                stts_2 = status;
}

printf("Producer process ended with status = %d\n", stts_1);
printf("Consumer process ended with status = %d\n", stts_2);

/* delete the IPC objects now that we're done. */
(void) semctl(semid, 0, IPC_RMID, 0);
(void) shmdt((char *) stack);
(void) shmctl(shmid, 0, IPC_RMID);
```

```
        exit(0);
}

/* Decrement semaphore count -- waiting if already 0 */
p(s)
int s;
{
        struct sembuf sops;
        int nsops = 1;

        sops.sem_num = s;
        sops.sem_flg = 0;
        sops.sem_op = -1;
        (void) semop(semid, &sops, nsops);
}

/* Increment semaphore count */
v(s)
int s;
{
        struct sembuf sops;
        int nsops = 1;

        sops.sem_num = s;
        sops.sem_flg = 0;
        sops.sem_op = 1;
        (void) semop(semid, &sops, nsops);
}

/* pop_item -- take item from the stack */
char *

pop_item()
{
        static char retstr[MESSAGE_LEN];
        if (stack->sp > 0) {
             strncpy(retstr, stack->stk[--stack->sp].message, MESSAGE_LEN);
             printf("pop: stack pointer = %d\n", stack->sp);
             return retstr;
        }
        return NULL;
}

/* push_item -- copy item into the next stack frame */
push_item(item)
char *item;
{
         strncpy(stack->stk[stack->sp++].message, item, MESSAGE_LEN);
         printf("push: stack pointer = %d\n", stack->sp);
}
```

Figure 8-26. Using Semaphore Calls

8.5.4 Message Calls

Messages provide a general method of communication between two processes. Using messages one process can pass information of any kind to another process. The information may be data that is produced by one process and used in another, or it could be flags that indicate when events occur. To use the message process, perform the following steps:

1. Use **ftok** to get a key assigned to a message queue.

2. Use the **msgget** call to get a message queue assigned to the processes.

3. Use the **msgsnd** call to send a message to a queue that is assigned to another process.

4. Use the **msgrcv** or **msgxrcv** call to receive a message from the message queue.

Use the following system calls to create and use message queues:

Call	Description
msgctl	Gets status, changes permissions, or removes a queue.
msgget	Gets message queue.
msgrcv	Receives a message.
msgsnd	Sends a message.
msgxrcv	Receives a message with additional information.

Include the following header files when using message queue calls:

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

Subtopics
8.5.4.1 Terms
8.5.4.2 General Operation
8.5.4.3 Controlling Bidirectional Queues
8.5.4.4 Using Message Queues
8.5.4.5 Example of Message Queue Calls

8.5.4.1 Terms

The use of message queue elements is similar in structure to the way the system creates and uses files. Defining the terms used for message queues with respect to the more familiar file terms provides a framework to build an understanding of message queues. Figure 8-27 shows the new terms and how they relate to terms used with files. The first three terms (key, msqid and permissions) pertain to how the kernel handles queues. The last term (queue name) is used in profiles (above the kernel).

Figure 8-27. Message Queue Terms

Term	**Definition**
key	The key is a unique identifier (of type **key_t**) that names the particular message queue. It is always associated with the message queue as long as the message queue exists. In this respect, it is similar to the *filename* of a file.
msqid	The msqid is an identifier assigned to the message queue for use within a particular process. It is similar in use to a *file descriptor* of a file.
permissions	The message queue structure also contains information that describes the access permissions for the message queue. These permissions are similar in function to the access permission bits for a file (owner, group and others).
queue name	The queue name is a 14-character, alphanumeric name that applies to a specific queue. The programmer or administrator chooses the name so that it does not conflict with other queue names on the system. Programs use the queue name to create or access the queue and get a key for that queue. Subsequent operations use the key value to refer to the queue.

In effect, message queues are a more general form of the **pipe** system call. Either method passes information between two processes. For message queues, however, you do not need to perform the steps of opening a pipe, forking, and then closing two of the ends of the pipe as described in "Example of a Pipe System Call" in topic 8.4.2.13. In fact, the two processes using message queues to communicate do not need to be created from the same ancestor process; they only need to cooperate by using the same name for the queue, and agreeing about what the messages mean.

8.5.4.2 General Operation

When a process gets a message queue, it uses an internal name (***key***) to apply to the queue. Any other programs that use that key can access that queue, subject to the read/write access permissions set up for the queue. The process can either send messages to the queue or receive messages from the queue (or both if it wants to send messages to itself).

After opening the queue, the process continues operating until it reaches the point that it needs input from the other cooperating process. The first process checks its message queue using the **msgrcv** call. Using the parameter, **msgtype**, in the **msgrcv** call, the process can specify which type of message it wants to receive. If a message that satisfies the request is not in the queue, the first process halts until something is put into the queue that does satisfy the request. If there is a message of the requested type in the queue, the system gives the process the first message of that type that was put into the queue (first-in-first-out).

Similarly, after opening the queue, the other process can send messages to the queue of the first process. If the queue is full, the system returns an error indication and the process must wait until the first process empties the queue enough to add the new message.

Because either of these wait conditions could halt the process indefinitely, the program should include a timeout loop to end the stalled condition.

Sending messages to a queue is completely independent from receiving messages from that queue. The amount of data that one process can put into the message queue of the other process depends on the queue size and the speed that the other process takes the data from the queue. More than one process can put messages into a queue. The receiving process must take them out of the queue in the order that they were put into the queue, modified only by selecting a message type.

The receiving process can also use the **msgxrcv** call instead of the **msgrcv** call to get messages. This call provides more information to the receiving process about the nature of the message.

8.5.4.3 Controlling Bidirectional Queues

When using a queue for a system service daemon, the requesting process can send a message to that queue and wait for a reply. In many cases, these queues can be set up for bidirectional communication; that is, both the daemon and the requesting process can get messages from the same queue. A bidirectional queue saves the overhead associated with creating and deleting a queue for the requesting process while it waits for the reply.

Use of a bidirectional queue requires cooperation among the processes. Routing the reply messages to the correct requesting process requires that each process generate and use a unique *mtype* parameter (message type) with all its messages to that queue. When sending messages to the queue, requesting processes send messages to the *mtype* of the daemon program. In each message they specify the *mtype* to which the daemon program must send the reply. The daemon program, being set up to receive messages of all types from the queue, gets all the messages. The daemon responds by generating a message of a type corresponding to the process for which the message is intended; the requesting process specifies in its **msgrcv** call that it will receive messages only of its unique type, and only receives the reply from the daemon that was intended for it.

To help assign unique *mtype* values, each message queue header includes a 32-bit value that contains the most recently used *mtype* value. A new command for **msgctl**, **IPC_MTYP**, returns the current value of the *mtype* of a queue. The *mtype* of the queue is incremented after it is returned to a process, but is not allowed to become negative.

Use the following guidelines when setting up bidirectional queues that use *mtype* for routing messages:

Reserve the *mtype* value of **0** for requests to the daemon program.

The requesting program uses **msgctl** with a command type of **IPC_MTYP** to get an *mtype* value from the queue header when the program gets a message queue ID. The kernel returns a unique *mtype* for each **msgctl** call.

The requesting program includes its *mtype* value as part of the data for each request sent to the queue.

When the daemon program replies, it uses the *mtype* value sent by the requesting program as the *mtype* of the reply message.

The requesting program waits for its reply by issuing a **msgrcv** that specifies only a message type that matches the *mtype* value sent to the queue.

8.5.4.4 Using Message Queues

The following sequence shows how to create and use a message queue:

1. Create a key to uniquely identify the message queue. Use the **ftok** subroutine to create the key. For example, to create a key **mykey** using a project ID of **X** contained in the **char** variable **proj** and a file name of **null_file**, use a statement like:

   ```
   mykey = ftok( null_file, proj );
   ```

2. Either:

   Create a new message queue with the **msgget** system call. For example, to create a message queue and assign the **msqid** to an integer variable **msg_qid**, use a statement like:

   ```
   msg_qid = msgget( mykey, IPC_CREAT );
   ```

   or

   Get a previously created message queue with the **msgget** system call. For example, to get a message queue that is already associated with the key **mykey** and assign the **msqid** to an integer variable **msg_qid**, use a statement like:

   ```
   msg_qid = msgget( mykey, IPC_ACCESS );
   ```

3. Use the queue to send or receive messages with other processes.

4. If the queue is no longer needed, eliminate it from the system using the **msgctl** system call:

   ```
   msgctl( msg_qid, IPC_RMID );
   ```

See *AIX Operating System Technical Reference* for specific information about parameters for the calls and subroutines.

8.5.4.5 Example of Message Queue Calls

The program on pages 8.5.4.5 through 8.5.4.5 shows the use of message queues in a simple producer-consumer relationship. One process produces an item for the other process and passes it to the other process on a message queue.

You can compile and run this program to see the effects of the system calls with the following command:

```
cc -o msgqtst msgqtst.c
```

```
/* msgq_test.c -- sample message queue test program */

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>

#define MSGSIZ        15   /* common size */
#define MSGRTYP       0    /* receive first message on queue */
#define MTYPE         1    /* common message type */

static char *msg[11] = {
                              "this is item 0", "this is item 1",
                              "this is item 2", "this is item 3",
                              "this is item 4", "this is item 5",
                              "this is item 6", "this is item 7",
                              "this is item 8", "this is item 9",
                              "this is item 10"
                              };
int msgqid;
struct msgbuf *msgp1;       /* pointer to producer mgsbuf */
struct msgbuf *msgp2;      /* pointer to comsumer msgbuf */

main()
{
      pid_t pid_1, pid_2, retpid, fork(), wait();
      key_t key, ftok();
      char *receive ();
      int send();
      int stts_1, stts_2, status;

      /* Allocate  msgbuf buffers */
      if ((msgp1 = (struct msgbuf *) malloc(MSGSIZ+sizeof(*msgp1))) == NULL)
            perror("malloc of msgp1");
            exit (1);
      }

      if ((msgp2 = (struct msgbuf * ) malloc(MSGSIZ+sizeof(*msgp2))) == NULL
            perror("malloc of msgp2");
            exit (1);
      }

      /* Create a message queue */
      key = ftok("msgq_test", 'm');
```

```
        if ((msgqid = msgget(key, IPC_CREAT ¦ S_IRUSR ¦ S_IWUSR)) < 0) {
                perror("msgq_test cannot open message queue");
                exit(1);
        }
        printf("msgqid = %d\n", msgqid);
        /* First child, the producer, will send messages */
        if ((pid_1 = fork()) == 0) {
                int i;

                  printf("Starting producer process (Child 1). \n");
                  for (i = 0; i <= 10; i++) {
                         printf("Producer: Sending messages number %d\n", i);
                         send(msg[i]);  /* produce item */
                  }
                  exit(0);
          }

          /* Second child, the consumer, will receive messages */
          if ((pid_2 = fork()) == 0) {

                  int i;
                  char *message;

                  printf("Starting consumer process (Child 2).\n");
                  for (i = 0; i <= 10; i++) {
                         message = receive ();   /* consume item */
                         printf("Consumer process: message number %d is: %s\n'
                                 i, message);
                  }
                  exit(0);
          }

          /* Wait for child processes to finish */
          stts_1 = stts_2 = -1;
          while ((retpid = wait(&status)) != -1) {
                  if (retpid == pid_1)
                         stts_1 = status;
                  else if (retpid == pid_2)
                         stts_2 = status;
          }

          printf("Producer process ended with status = %d\n", stts_1);
          printf("Consumer process ended with status = %d\n", stts_2);

          /* delete the message queue now that we're done */
          (void) msgctl(msgqid, IPC_RMID, 0);

          exit(0);
}

send(item)
char *item;
{
          msgp1->mtype =MTYPE;
          strncpy(msgp1->mtext, item, MSGSIZ);
          if (msgsnd(msgqid, msgp1, MSGSIZ, 0) == -1)
                    perror("msgsnd");
}
char *
receive()
```

```
{
        if (msgrcv(msgqid, msgp2, MSGSIZ, MSGRTYP, 0) == -1)
                perror("msgrcv");
        return msgp2->mtext;
}
```

8.5.5 TCP/IP Socket Communication

For detailed information on TCP/IP Socket Communication, refer to *AIX TCP/IP User's Guide*.

8.5.6 Shared Memory Calls

The ***shared memory*** calls set aside an area of memory that cooperating processes can access. This area can serve as a large pool for exchanging data among the processes. The shared memory calls do not provide locks or access control among the processes. Therefore, processes using the shared memory area must set up a signal or semaphore control method to prevent access conflicts and to keep one process from changing data that another process is using. Use shared memory when the amount of data to be exchanged between processes is too large to transfer with messages, or when many processes maintain a common large data base.

Use the following calls to create and use shared memory segments from a program:

Call	Description
shmctl	Controls shared memory operations.
shmget	Gets or creates a shared memory segment.
shmat	Attaches a shared memory segment to a process.
shmdt	Detaches a shared memory segment from a process.
disclaim	Removes mapping from a specified address range within a shared memory segment.

In addition, the **ftok** subroutine provides the ***key*** that the **shmget** call uses to create the shared segment. Include the following header files when using shared memory calls:

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

Subtopics
8.5.6.1 Terms
8.5.6.2 Using Shared Segments

8.5.6.1 Terms

The use of shared memory segments is similar in structure to the way the system creates and uses files. Defining the terms used for shared memory with respect to the more familiar file terms provides a framework to build an understanding of shared memory. Figure 8-28 shows the new terms and how they relate to terms used with files.

Figure 8-28. Shared Memory Terms

Term	**Definition**
key	The ***key*** is a unique identifier that names the particular shared segment. It is always associated with the shared segment as long as the shared segment exists. In this respect it is similar to the *filename* of a file.
shmid	The ***shmid*** is an identifier assigned to the shared segment for use within a particular process. It is similar in use to a *file descriptor* of a file.
attach	A process must ***attach*** a shared segment to use the shared segment. Attaching a shared segment is similar to opening a file.
detach	A process must ***detach*** a shared segment once it is finished with the shared segment. Detaching a shared segment is similar to closing a file.

8.5.6.2 Using Shared Segments

The following sequence describes the life cycle of a shared segment from initial creation to final removal from the system:

1. Create a key to uniquely identify the shared segment. Use the **ftok** subroutine to create the key. For example, to create a key **mykey** using a project ID of **R** contained in the variable **proj** (type **char**) and a file name of **null_file**, use a statement like:

   ```
   mykey = ftok( null_file, proj );
   ```

2. Either:

   Create the shared memory segment with the **shmget** system call. For example, to create a shared segment that contains 4096 bytes and assign the **shmid** to an integer variable **mem_id**, use a statement like:

   ```
   mem_id = shmget( mykey, 4096, IPC_CREATE or 0666 );
   ```

   or

   Get a previously created shared segment with the **shmget** system call. For example, to get a shared segment that is already associated with the key **mykey** and assign the **shmid** to an integer variable **mem_id**, use a statement like:

   ```
   mem_id = shmget( mykey, 4096, IPC_ACCESS );
   ```

3. Attach the shared segment to the process with the **shmat** system call. For example, to attach the previously created segment, use a statement like:

   ```
   ptr = shmat( mem_id );
   ```

   In this example, the variable **ptr** is a pointer to a structure that defines the fields in the shared segment. Use this template structure to store and retrieve data in the shared segment. This template should be the same for all processes using the segment.

4. Work with the data in the segment using the template structure.

5. Detach from the segment using the **shmdt** system call:

   ```
   shmdt( mem_id );
   ```

6. If the shared segment is no longer needed, remove it from the system using the **shmctl** system call:

   ```
   shmctl( mem_id, IPC_RMID, ptr ) ;
   ```

See *AIX Operating System Technical Reference* for specific information about parameters for the calls and subroutines. You can also use the commands **ipcs** to get information about a segment and **ipcrm** to remove a segment. See *AIX Operating System Commands Reference* for information about these commands.

8.5.7 Memory Management Calls

Use the following calls to control a program's use of memory during execution:

Call	Description
brk, sbrk	Change data segment space allocation.

8.6 File System Calls

The system provides calls to create files, move data into and out of files, close files, and describe the restrictions and structure of the file system. Because the system treats input and output to all devices the same as input and output to files, you can use many of the ***file system*** calls for control of devices in the system also. The system calls do not provide the data formatting and housekeeping services that the C library subroutines for input and output do. See Chapter 7, "Using the Subroutine Libraries" for information about the library calls.

Subtopics
8.6.1 Data Handling Calls
8.6.2 File Maintenance Calls

8.6.1 Data Handling Calls

Use the following system calls to control data handling in the system. These calls create files, open and close files, and move data into and out of them.

Call	Description
fcommit	Makes permanent all changes to a file since the last commit operation.
fsync	Equivalent to **fcommit**.
close	Closes a file descriptor.
closex	Closes a file descriptor and uses an extra parameter for communication with the device driver.
creat	Creates a new file or rewrite an existing one.
dup	Duplicates an open file descriptor.
ioctl	Controls device.
ioctlx	Controls device and uses an extra parameter for communication with the device driver.
lseek	Moves read/write file pointer.
mknod	Makes a special file that provides an interface to a device for input and output.
mknodx	Make a special file that provides an interface to a device for input and output and uses an extra parameter to specify the cluster site to which the device is attached.
open	Opens a file or device for reading or writing.
openx	Opens a file or device for reading or writing and uses an extra parameter for communication with the device driver or controlling how directories appear.
read	Reads from a file or device.
readx	Reads from a file or device and uses an extra parameter for communication with the device driver or for controlling how directories appear.
fabort	Aborts file changes that have occurred since the last commit operation.
write	Writes on a file or device.
writex	Writes on a file or device and uses an extra parameter for communication with the device driver.

Subtopics
8.6.1.1 Using Files
8.6.1.2 File Descriptors
8.6.1.3 Opening and Closing Files
8.6.1.4 Random Access to Files

8.6.1.1 Using Files

When transferring data on the system, ensure that the files are created, opened and closed at the proper times so that data is not lost. The following sequence of events describes the actions to perform when using files for data storage:

1. Use the **creat** system call to create the file **or** Use the **open** system call to open the file if it already exists (the **creat** call leaves the file open).

2. Use the **write** and **read** system calls to transfer data into and out of the file.

3. Use the **close** system call to close the file.

8.6.1.2 File Descriptors

The system performs all input and output by reading or writing files. It uses ***special files*** to form the interface between a device and the operating system. When you open a file, the system checks to see if you can access it. If you have access to the file, the system returns a small positive integer called a ***file descriptor***. The system uses this file descriptor instead of the name to identify the file. Therefore, programs must use the file descriptor when doing input and output with system calls.

The system assigns the file descriptor number on an *as available* basis. Normally, when a program begins, the system establishes three numbers for special functions:

0 ***standard input***: This file normally handles input from the keyboard of the terminal.

1 ***standard output***: This file normally handles output to the terminal screen.

2 ***standard error***: This file normally handles error messages to the terminal screen.

These file descriptors are normally always open, so that a program can get input from the standard input and send output to standard output or error without opening a file.

8.6.1.3 Opening and Closing Files

To use input and output other than **stdin**, **stdout**, or **stderr**, either **open** or **creat** a file. The **open** system call sets up an existing file for access, and returns a file descriptor to the calling program:

```
int fildes;
fildes = open (filename,oflag[,mode]);
```

In this call, the parameters have the following meaning:

filename The character string that corresponds to the external file name of the file to be opened.

oflag A flag that indicates the conditions of the access for the file.

mode The optional access permission bits for the file. These bits are used only if the file must first be created before it can be opened. See the **chmod** command in *AIX Operating System Commands Reference* for an explanation of the access permission bits.

If an error occurs, **open** returns **-1** as the file descriptor. Trying to open a file that does not exist is an error.

To create a new file, use the **creat** system call:

```
fildes = creat(filename,mode);
```

In this call, the parameters have the following meaning:

filename The character string that corresponds to the external file name of the file to be opened.

mode The access permission bits for the file to be created as described for the **chmod** command in *AIX Operating System Commands Reference*.

The **creat** call returns **-1** as the file descriptor if it cannot create the file. If the file exists, this call truncates that file to zero length and returns the file descriptor for that file.

To free up a file descriptor for use with another file, use the **close** system call when access to the file is complete:

```
close(fildes);
```

When the program stops using an **exit** call, or a **return** from **main**, the system closes all file descriptors associated with the program.

8.6.1.4 Random Access to Files

Normal access to files is sequential. Each **read** or **write** occurs in the file position directly following the previous operation. To perform random access I/O, use the **lseek** system call to move around in the file. This system call does not read or write to the file, it only changes the position where the next **read** or **write** will occur. The format of this call is:

```
lseek(fildes, offset, whence);
```

In this call, the parameters have the following meaning:

fildes The file descriptor for the file.

offset The number of bytes to move, or an absolute address as specified by the *whence* parameter.

whence Determines how to use *offset* to move in the file:

SEEK_SET Moves to the address contained in *offset*.

SEEK_CUR Moves to the address that is the current location plus the value contained in *offset*.

SEEK_END Moves to the address that is the number of bytes contained in *offset* plus the address of the end of the file.

The symbolic names described above are defined in **/usr/include/unistd.h**.

For example, to append to a file when it is not positioned at the end, seek to the end before writing:

```
lseek(fildes, 0, SEEK_END);
```

To get back to the beginning of the file:

```
lseek(fildes, 0, SEEK_SET);
```

To create ***sparse files***, or files with ***holes*** in them to allow for relative record access within the file, create a new file and then use the **lseek** call to move to selected places in the file before writing. The spaces between the data (***holes***) become part of the file, but do not take up disk space until they are actually filled with data.

8.6.1.5 Reading and Writing to a File

The **read** and the **write** system calls perform input and output for the system. These calls require three arguments:

File descriptor The integer assigned to the file involved in the read or write.

Buffer An area in the program that supplies or receives the data.

Byte count The number of bytes to be transferred.

Each call returns the number of bytes actually transferred. For a **read**, this number may be less than the number requested in the byte count parameter. For a **write**, if this number is not the number requested, an error occurred. A returned value of **0** indicates the end of file; a returned value of **-1** indicates an error occurred during the operation.

The number of bytes to transfer is the programmer's choice. Some useful values are:

1 One character at a time, or *unbuffered* transfer.

512 The block size for many peripheral devices.

4096 The internal block size for the operating system. This size, or a multiple of this size, is efficient for normal operations.

8.6.1.6 Using the Extended Calls

The system provides extended versions of the following system calls:

closex
ioctlx
openx
readx
writex.

These calls perform the same functions as their original versions, but they also provide an extra parameter to pass information to the device driver. How the parameter is used depends on the device driver with which the calls are used. The parameter can be used either as a value or a pointer to a buffer area containing additional information. Use them only with device drivers that understand the additional information.

8.6.1.7 Committing File Changes

In most cases, closing the file after writing to it is enough to ensure that changes to it are permanent. The call **fcommit(fildes)** incorporates all data changes specified by the given file descriptor into a new version of the file. The file remains open after the commit; a subsequent call of the form **fabort(fildes)** will roll the file back to the most recent commit checkpoint.

A program which uses **fcommit** and **fabort** to control when changes are incorporated in a file should first open the file with the O_DEFERC open flag. This flag prevents the system from automatically committing the file at intermediate times. **fsync** is equivalent to **fcommit** and is included for system compatibility. Refer to the discussion of **fsync** and **fabort** in the *AIX Operating System Technical Reference*.

8.6.2 File Maintenance Calls

Use the file maintenance calls for programs that change protection of files in the file system, access many different files, or provide control of files for the user of the program. Many of these calls are the base for the system commands that have similar names. You can, however, use these calls to write new commands or utilities to help in the program development process, or to include in an application program. The file maintenance calls on the system include:

Call	Description
access	Determines accessibility of a file.
chdir	Changes working directory.
chhidden	Marks an existing directory as a hidden directory.
chmod	Changes mode of a file.
chown	Changes owner and group of a file.
chroot	Changes root directory.
dirstat	Returns statistical information on a block of files in a directory.
fchmod	Changes mode of an open file.
fchown	Changes owner and group of an open file.
fclear	Clears space in a file.
fcntl	Controls file operations.
fstat, fstatx	Gets attributes of an open file.
link	Links to a file.
lockf	Locks a region of a file, or provides exclusive regions in a file.
mount	Mounts a file system.
mkdir	Makes a directory
readlink	Reads value of symbolic link.
rename	Renames a file.
rmdir	Removes a directory.
rmslink	Removes a symbolic link.
select	Control multiple open files at once.
stat, statx	Gets attributes of a file.

symlink	Makes a symbolic link to a file.
sync	Updates superblock.
truncate, ftruncate	Makes file shorter.
umask	Sets and gets file creation mask.
unlink	Removes a directory entry.

Call	**Description**
umount	Unmounts a file system.
ustat,dustat	Gets file system statistics.
utime	Sets file access and modification times.

Subtopics
8.6.2.1 Symbolic Links

8.6.2.1 Symbolic Links

Ordinarily, there is a one-to-one correspondence between a file system pathname and the file system object to which it is bound. A **link** is a mechanism by which more than one name is bound to the same object. There are two types of links:

hard links

symbolic links.

The **link** and **symlink** system calls bind new path names to existing objects, creating hard link and symbolic links, respectively. These links can later be removed by the **unlink** and **rmslink** system calls. Each type of link has certain advantages over the other. For additional information regarding hard links and symbolic links, refer to *Using the AIX Operating System*.

8.7 Time System Calls

The system provides the following calls to set the system time and to find out what the system time is. See the description for the **ctime** library routine in *AIX Operating System Technical Reference* to get formatted time data from the system.

Call	Description
stime	Sets time.
time	Gets time.

Both calls use a value of time that is the number of seconds since 00:00:00 Greenwich Mean Time (GMT) on January 1, 1970. Therefore, the program must be able to calculate the date using that starting date and the elapsed time value used by the time calls, total seconds. For convenience, some of the common time units converted to seconds are:

Unit	Value in Seconds
minute	60
hour	3600
day	86,400
week	604,800
month	2,419,200 (28 days)
	2,505,600 (29 days)
	2,592,000 (30 days)
	2,678,400 (31 days)
year	31,536,000 (365 days)
	31,622,400 (366 days)

Subtopics

8.7.1 Using File Locking

AIX Operating System allows many processes to synchronize simultaneous access to a file through the use of the **fcntl** and **lockf** system calls. These system calls allow a program to lock and unlock portions of an open file. The program can use either a read lock or a write lock.

A ***read lock*** prevents any other process from setting a write lock on any portion of the protected area. When a read lock is set on a segment of a file, other processes can also set read locks on that segment or a portion of it. The file descriptor on which a read lock is being placed must have been opened with read access. Use the **fcntl** system call to set a read lock on a file.

A ***write lock*** prevents any other process from setting a read lock or a write lock on any portion of the protected area. Only one write lock and no read locks can exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access. Use the **fcntl** system call or the **lockf** system call to set write locks.

All locks associated with a file for a given process are removed when the file descriptor for that file is closed by that process or when the process holding the file descriptor ends. Locks are not passed to a child process after executing a **fork** system call.

File locks can operate as either ***enforced*** or ***advisory*** locks. Enforced locks prevent other processes from accessing the locked area of the file. Advisory locks are not enforced by the operating system and require cooperating processes that check the lock bits of files to ensure data integrity. To select enforced locking, the **S_ENFMT** code must be set in the access permission bits (or mode) of the file. Using the **chmod** system call on a file that has locks can change the type of lock between forced and advisory. Otherwise, locking is advisory. Thus, a particular file can have advisory or enforced locks, but not both. Use advisory locks instead of enforced locks whenever possible.

A process cannot use a **write** system call on any region of a file that is protected by an enforced read lock or an enforced write lock that is held by another process. A process cannot use a **read** system call on any region of a file that is protected by an enforced write lock that is held by another process. In addition, the **creat**, **open**, **truncate**, **ftruncate**, **fabort**, and **fclear** system calls cannot truncate or change a file protected by any enforced lock that is held by another process.

When an area of a file is protected by an enforced lock and another process attempts to access that area of the file, the attempt is not successful but the result varies with other process conditions. A **read** or **write** system call that is blocked may either sleep until the area is unlocked or, if the **O_NDELAY** flag for that file descriptor is set, return with an error. However, if the system detects that sleeping would cause deadlock, then the system call fails with **errno** set to **EDEADLK**. If another process attempts to truncate the file with either the **creat** or **open** system call, that system call fails with **errno** set to **EACCES**.

The **read**, **write**, **creat** and **open** system calls are not affected by advisory locking. Processes must voluntarily call **lockf** to make advisory locks effective.

File and record locking helps preserve data integrity when multiple

processes are using the same file. When using Transparent Computing Facility (TCF) or Network File Service (NFS), this protection applies to both local and remote files. Processes that use record and file locking in a local area network must be careful not to introduce permanent deadlocks when they wait for locks. Before waiting for a lock, these processes should set a timer so that they will be awakened by a signal if there is the possibility of a deadlock.

Subtopics
8.7.1.1 File Locking Example Program

8.7.1.1 File Locking Example Program

The following example program shows how to use locking system calls. Compile the program to an executable file named **locktest** with the following command:

```
cc locktest.c -o locktest
```

You can then run the program by entering the command:

```
locktest filename
```

In this format, *filename* is the name of a temporary file that the program creates. Ensure that this name is not the name of any file in the current directory. For example, to start the program and allow it to create a file named **testfile**, enter the command:

```
locktest testfile
```

When the program runs, it:

1. Creates a signal handler to catch **SIGALRM** if it occurs.
2. Creates the test file with enforced locking enabled using the **open** system call.
3. Uses the **lockf** system call to enforce lock the whole file from the parent process.
4. Uses the **fork** system call to create a child process.
5. Writes 15 records to the file from the parent process while the child process tries to lock one record in the file.
6. The child process sleeps until the parent process unlocks the file.
7. The parent process unlocks the file.
8. The child process wakes up and completes its record lock.
9. The child process reads one record from the file.
10. The child process removes its record lock from the file and then exits.
11. The parent process detects the end of the child process and exits.

Figure 8-29 shows the program listing for **locktest.c**. Figure 8-30 shows the output produced when the program runs. When the program completes successfully, it creates a test file that contains a string of test patterns containing 40 characters and a new line. When you are done with the program, erase the test file.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/lockf.h>
```

```
#include <sys/stat.h>
#include <sys/signal.h>
#include <errno.h>

#define  TIMEOUT        10
#define  RECORDSIZE     40
#define  RECORD(x)      (((x)-1)*RECORDSIZE)


static char record[] = "This string has 40 characters w/newline\n";
void
salrm()         /* signal handler for receiving SIGALRM */
{
   printf("SIGALRM signal received.\n");
}

main(argc, argv)
int argc;
char **argv;


{
   int fildes;             /* file descriptor */
   int i, rc;              /* for loop index and return code */
   char buf[RECORDSIZE];   /* read buffer */
   static struct sigaction alrmvec;
   alrmvec.sa_handler = salrm;
   alrmvec.sa_flags = 0;
   sigemptyset (&alrmvec.sa_mask);


   if( argc != 2 )
   {
      usage();
      exit(1);
   }


   if( sigaction(SIGALRM, &alrmvec, NULL) == -1 )
   {
      perror("sigaction(2) failed ");
      exit(1);
   }


   if( (fildes=open(argv[1], O_RDWR | O_CREAT | O_TRUNC,
       S_ENFMT | S_IRUSR | S_IWUSR ))== -1 )
   {
      perror("open(2) failed ");
      exit(1);
   }


   alarm(TIMEOUT);
   printf("Parent process tries to lock the entire file.\n" );
   if( lockf(fildes, F_LOCK, 0) == -1 )
   {
      perror("parent process lockf(2) failed ");
      exit(1);
   }
```

```
alarm(0);
printf("Fork the child process.\n" );
fflush (stdout);
switch(fork())


{
case -1:                              /* fork error */
   perror("fork(2) failed ");
   exit(1);


case 0:                               /* child process */
   printf("Child process tries to lock the 5th record in the file.\n" );
   lseek(fildes, RECORD(5), SEEK_SET);
   alarm(TIMEOUT);
   if( lockf(fildes, F_LOCK, RECORDSIZE) == -1 )


   {
      perror("child process lockf(2) failed ");
      exit();
   }


   alarm(0);
   if( (rc = read(fildes, buf, RECORDSIZE)) == -1 )
   {
      perror("child process read(2) failed ");
      exit();
   }


   printf("Child process read record 5 from the file.\n" );
   printf("Child process tries to unlock the 5th record in the file.\n" )
   lseek(fildes, RECORD(5), SEEK_SET);

   if( lockf(fildes, F_ULOCK, RECORDSIZE) == -1 )
   {
      perror("child process lockf(2) failed ");
      exit();
   }

   close(fildes);
   exit(0);

default:                              /* parent process */
   for( i=1; i<=15; i++ )
   {
      if( (rc = write(fildes, record, RECORDSIZE)) == -1 )
      {
          perror("parent process write(2) failed ");
          exit();
      }

      printf("Parent process wrote record %d to the file.\n" , i);
   }
```

```
        printf("Parent process tries to unlock the entire file.\n" );
        lseek(fildes,0,SEEK_SET);

        if( lockf(fildes, F_ULOCK, 0) == -1 )
        {
           perror("parent process lockf(2) failed ");
           exit(1);
        }

        printf("Parent process waits for the child to end.\n" );
        wait(0);
        printf("Child process has ended; Parent process ends.\n" );
        close(fildes);
        exit(0);
     }
  }
  usage()
  {
     printf("usage: locktest <file>\n");
  }
```

Figure 8-29. Example Program Listing for locktest.c

```
  Parent process tries to lock the entire file.
  Fork the child process.
  Child process tries to lock the 5th record in the file.
  Parent process wrote record 1 to the file.
  Parent process wrote record 2 to the file.
  Parent process wrote record 3 to the file.
  Parent process wrote record 4 to the file.
  Parent process wrote record 5 to the file.
  Parent process wrote record 6 to the file.
  Parent process wrote record 7 to the file.
  Parent process wrote record 8 to the file.
  Parent process wrote record 9 to the file.
  Parent process wrote record 10 to the file.
  Parent process wrote record 11 to the file.
  Parent process wrote record 12 to the file.
  Parent process wrote record 13 to the file.
  Parent process wrote record 14 to the file.
  Parent process wrote record 15 to the file.
  Parent process tries to unlock the entire file.
  Child process read record 5 from the file.
  Child process tries to unlock the 5th record in the file.
  Parent process waits for the child to end.
  Child process has ended; Parent process ends.
```

Figure 8-30. Output from Example Program locktest.c

9.0 Chapter 9. Controlling the Terminal Screen

Subtopics

9.1 CONTENTS

9.2 About This Chapter

This chapter discusses the **Extended curses** library, which contains routines to support input and output to the terminal screen. It also contains information on using the library routines for panels and panes. In addition, it describes attributes that a program can use on a terminal.

9.3 Extended Curses

The system contains two libraries of routines to support input and output to the terminal screen. These libraries are:

curses A set of screen control routines. This library is included for compatibility with existing application programs.

Extended curses An enhancement to the **curses** set of routines for IBM PS/2 that provides extended function for:

Expanded character set
Color
Multiple character attributes
Error detection and handling
Efficient handling of a window-oriented screen presentation, including:
- Window stacking and layers
- Linked scrolling of windows
- Scrolling data in windows that are partially covered
- Automatic tracking of active panes.

2-byte characters for international character support
locator input support.

Use these routines for new program development or to increase the function of existing programs.

This chapter discusses only the **Extended curses** library. Information about both libraries is in *AIX Operating System Technical Reference*.

The **Extended curses** library contains a set of C language routines that:

Updates a scree

Gets input from the terminal in a screen-oriented fashio

Moves the cursor from one point to another independent of other scree activities

Creates and manages a screen containing windows, panels and panes

The routines do the most common type of terminal-dependent functions. The routines use the file **/usr/lib/terminfo** to describe what the terminal can do.

The routines are in the following categories:

Screen updatin

Screen updating with user inpu

Cursor motion optimization

You can use motion optimization by itself. You can use screen updating and input without knowing about either motion optimization or the data.

Subtopics
9.3.1 New Terms
9.3.2 What You Need
9.3.3 Using the Screen Update Routines
9.3.4 What the Screen Looks Like
9.3.5 Function Names
9.3.6 Variables

9.3.1 New Terms

The **Extended curses** routines use the concepts and terms listed in Figure 9-1.

Figure 9-1. Terms

Term	Definition
terminal	Sometimes called ***terminal screen***, the ***terminal*** is a memory image of what the terminal screen currently looks like. This is a special *screen*.
screen	A ***screen*** is a special type of window that is as large as the terminal screen. You can define screens for a program. The **Extended curses** routines define two screens for their use: **stdscr** The ***standard screen*** is a memory image of the screen that the routines make changes to. **curscr** The ***current screen*** is the actual image that is currently on the terminal.
window	A memory image of what a section of the terminal screen looks like at some point in time. A window can be either the entire terminal screen, or any smaller portion down to a single character.
Presentation Space	The data and attribute array associated with a window.
Pane	An area of the display that shows all or a part of the data contained in a presentation space associated with that pane. A pane is a subdivision of a panel.
Panel	A rectangular area on the display consisting of one or more panes that a program can treat as a unit. That is, the panes in a panel are displayed together, erased together and represent a unit to the operator. The routines stack or overlap panels on the screen, and remember the order of the stack and the contents of each panel.
Field	An area in a presentation space where the program can accept operator input.
extended character	A character from a code page that is represented in 2 bytes.

9.3.2 What You Need

To use the library, define the types and variables that the routines use. The file **cur01.h** contains all of the definitions that are needed for the library routines for most common uses. Include this file in the program by putting the statement:

```
#include <cur01.h>
```

at the top of the program source. When using the library routines for panel and pane management (those routines begin with the letters **ec**), use the statement:

```
#include <cur05.h>
```

in the program source to include a larger set of definitions.

Use an additional header file **cur00.h** in the program if the program uses the global variables defined to represent the information taken from the terminal file. This header file also contains include statements for the following header files:

- stdio.
- sgtty.
- cur01.

You do not need to include those files separately in the program.

To compile a program with the **cc** command, specify the two additional libraries shown in the following example on the command line. This example compiles the program, **myprog.c**, with the linked output going to **a.out**, by using the following command:

```
cc myprog.c -lcur -lcurses
```

See Chapter 7, "Using the Subroutine Libraries" for information about using libraries in a program.

9.3.3 Using the Screen Update Routines

To update the screen, the routines must know what the screen currently looks like and what it should be changed to. The routines define a data type, WINDOW, to hold this information. This data type is a structure that describes a window image to the routines, including the starting position on the screen (the (line, col) coordinates of the upper left corner) and size. See Appendix A, "Extended curses Structures" for a definition of the WINDOW structure.

A window is like an array of characters on which to make changes. Using the window, a program builds and stores an image of a portion of the terminal that it later transfers to the actual screen. When the window is complete, use one of the following routines to transfer the window to the terminal:

refresh Transfers the contents of **stdscr** to the terminal.

wrefresh Transfers the contents of a named window (not **stdscr**) to the terminal.

ecrfpl Transfers the contents of a named panel to the terminal.

ecrfpn Transfers the contents of a named pane to the terminal.

This two-step process maintains several different copies of a window in memory and selects the proper one to display at any time. In addition, the program can change the contents of the screen in any order. When it has made all of the changes, the library routines update the terminal in an efficient manner.

9.3.4 What the Screen Looks Like

The screen is a matrix of character positions that can contain any character from the character set (see Appendix C, "ASCII Characters") that can be displayed. Do not use control characters except when the descriptions of the library indicate that you can. The actual dimensions of the matrix are different for each type of terminal. These dimensions are defined when the **initscr** routine calls the **terminfo** initialization subroutine, **setupterm**. For more information about **setupterm**, see *AIX Operating System Technical Reference*. However, the routines enforce the following limits on the terminal:

Coordinate	**Description**
lines	If the terminal specification defines less than 5 lines, the routines use a value of 24 lines.
columns	If the terminal specification defines less than 5 columns, the routines use a value of 80 columns.

Line 0 is at the top of the screen. Line values in the routine syntax are represented by **line**. Column 0 is at the left side of the screen. Column values in the routine syntax are represented by **col**. When used in calls to the library routines, the *line* value comes first.

```
move(line, col);
```

Figure 9-2 shows the coordinate boundaries for a screen with 48 lines and 1000 columns.

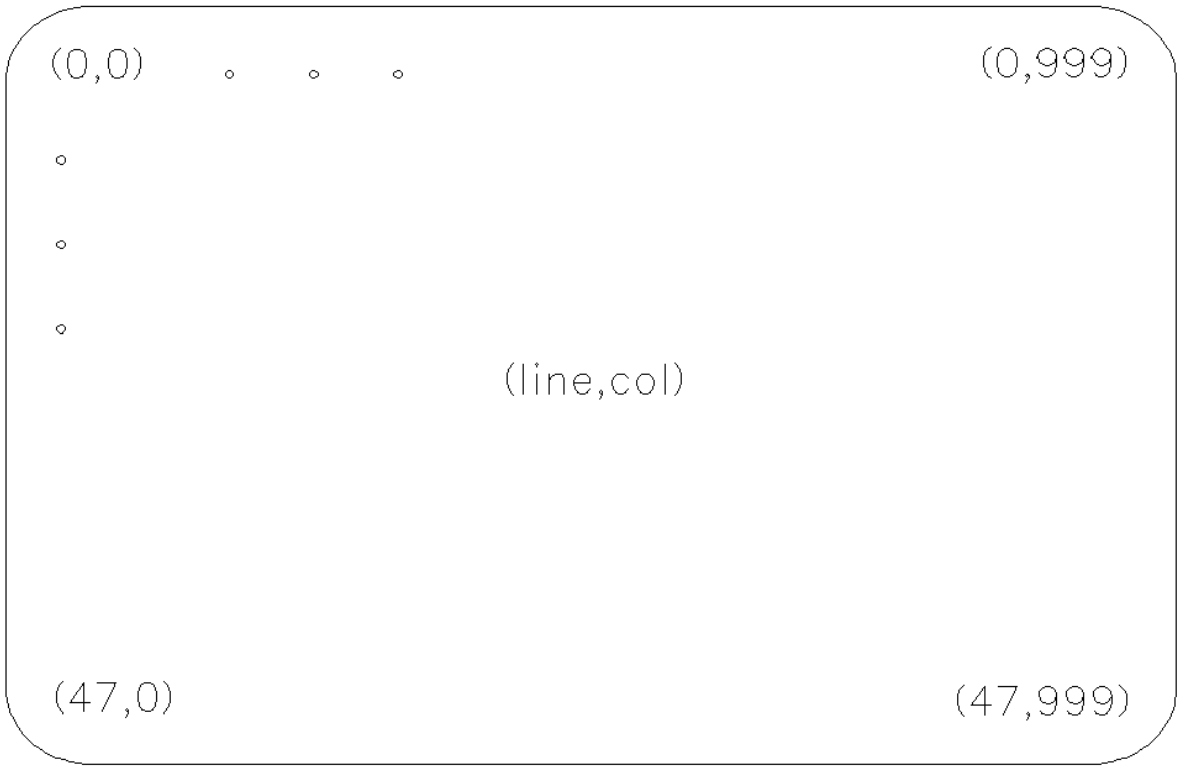


Figure 9-2. Screen Coordinate Boundaries

9.3.5 Function Names

Functions that change the contents of a specified window other than **stdscr** usually begin with the letter **w**, indicating an operation for a specific window. Deleting the leading **w** provides the name of the same function that uses **stdscr**. For example, the function **addch** adds a character to **stdscr**, but the function **waddch** adds a character to a specified window. If a function does not have a form that operates only on **stdscr**, the function does not have a form that begins with the letter **w**. Always indicate a window name when using these functions.

Use the routines **move** and **wmove** to change the current (line, col) coordinates from one point to another. To move and then write to or read from the new position, use the following shorthand method for most routines:

1. Add the letters **mv** to the front of the routine name.

2. The first two arguments of the routine must be the (line, col) coordinates of the destination of the move.

For example, the following call sequence:

```
move(line, col);
addch(ch);
   .
   .
   .
wmove(win, line, col);
waddch(win, ch);
```

is the same as the following call sequence:

```
mvaddch(line, col, ch);
   .
   .
   .
mvwaddch(win, line, col, ch);
```

Note that the window description pointer **win** comes before the added (line, col) coordinates.

9.3.6 Variables

The following system variables defined in the header files describe the terminal environment. Use these variables in a program.

Name	Type	Description
My_term	bool	If this value is TRUE, the routines use the terminal type specified by **Def_term** as the terminal type being used. If this value is FALSE, the routines first check the terminal type specified in the **$TERM** for the system environment. If **$TERM** is not specified, the routines use the value in **Def_term**.
Def_term	char*	Default terminal type if **$TERM** is not specified in the system environment.
COLS	int	Number of columns on the terminal.
ERR	int	Flag that the routines return when a failure occurs.
LINES	int	Number of lines on the terminal.
OK	int	Flag that the routines return when the function completes successfully.
curscr	WINDOW*	Current version of the terminal screen.
stdscr	WINDOW*	Standard screen.

The routines also define the following **#define** constants and types:

bool A type of *boolean* used as:

```
bool doneit;    /* defines variable doneit */
```

reg A type with storage class *register* used as:

```
reg int i;      /* defines i */
```

FALSE The value of boolean false (0).

TRUE The value of boolean true (1).

NLSCHAR A data type defined as unsigned short.

ATTR A data type defined as unsigned short.

9.4 Using the Library Routines

The following paragraphs outline the steps to follow when building a program to use these routines. The description uses routines that change **stdscr**, but the same concepts work with any window when using the **w** form of the routine as described in "Function Names" in topic 9.3.5. See *AIX Operating System Technical Reference* for complete descriptions of these library routines, and their **w** forms.

Subtopics

9.4.1 Setting Up the Environment

To use these routines, the program must set up the operating conditions for the program. Perform the following actions in the program, if they apply, in the order that they appear in the following procedure:

1. Perform all necessary actions to load the program and make sure that it is operating successfully.

2. To change the defined size of the terminal, set the variables **LINES** and **COLS** to new values.

3. Use the routine **initscr** to get information about terminal characteristics, and to allocate memory for **stdscr** and **curscr**. Call **initscr** before calling any routines that affect windows. If the program uses a window routine before **initscr**, the program will not run.

4. Check the value that **initscr** returns to see if the screen setup was successful. If this value is **0** (**FALSE** or **ERR**), then **initscr** could not get enough memory for the needed windows.

5. Use any needed terminal status changing routine, such as **nl** or **crmode**.

6. Create any new windows with the **newwin** or **subwin** routine.

7. Create panels using **ecbpls**, **ecbpns**, **ecdvpl** or **ecdfpl**.

8. Define or change the characteristics of the windows as needed. For example, the routine **scrollok** allows the window to scroll, or the routine **leaveok** leaves the cursor at the position of the last change.

The program can now work with the windows that it has defined. When the program is done, use the routine **endwin** to clean up before exiting the program. This routine restores terminal modes to what they were when the program first started.

9.4.2 Writing to a Window

Use the following functions to change the contents of a window. Refer to *AIX Operating System Technical Reference* for complete information about each routine.

Routine **Description**

addch(c)
Adds the character **c** on the window at the current (line, col) coordinates.

waddfld (win, string, length, numlines, numcols, mode, xc)
Adds a string to the window within a specified area (field) starting at the current coordinates.

addstr(str)
Adds the string pointed to by **str** on the window at the current (line, col) coordinates.

box(win, vert, hor)
Draws a box around the window using **vert** as the character for drawing the vertical sides, and **hor** for drawing the horizontal lines. See also **fullbox** and **cbox**.

cbox(win)
Draws a box around the window using the box characters defined in **/usr/lib/terminfo** (BX[ ]). If no box characters are defined, it uses **or** for vertical lines, **-** for horizontal lines, and **+** for corners. See also **box** and **fullbox**.

chgat(num_chars, mode)
Changes the attributes of the next **num_chars** characters, starting at the current (line, col) coordinates to the attribute(s) specified by **mode** (one or more of the attributes defined in "Display Attributes" in topic 9.6). See also **pchgat**.

clear ()
Resets the entire window to blanks.

clearok(scr, boolf)
Sets the clear flag for the screen **scr** to the value of **boolf**.

clrtobot ()
Clears the window from the current (line, col) coordinates to the bottom.

clrtoeol ()
Clears the window from the current (line, col) coordinates to the end of the line.

colorend ()
Returns the terminal to *normal* attributes following a **colorout** call.

colorout(mode)
Sets the current standout bit pattern (_csbp in the **window** structure) to the value of **mode** (one or more of the attributes defined in "Display Attributes" in topic 9.6) and turns on **_STANDOUT**. All characters following this call are displayed

with **mode** as the attribute. Multiple attributes are set by OR-ing them together.

delch ()
Deletes the character at the current (line, col) coordinates.

Routine Description

deleteln ()
Deletes the current line.

ecactp (pane, boolt)
Specifies the active pane.

ecshpl (panel)
Shows a specified panel by bringing it to the top of the stack of panels.

ecrfpl (panel)
Refreshes the panel on the display.

ecrfpn (pane)
Refreshes the pane on the display.

ecrmpl (panel)
Removes a panel from the display.

ecscpn (pane, numlines, numcols)
Scrolls a specified pane.

erase ()
Erases the window to blanks without setting the clear flag. See also **perase**.

fullbox(win, vert, hor, topl, topr, botl, botr)
Draws a box around the window using **vert** as the character for vertical sides, **hor** for horizontal, and **topl, topr, botl** and **botr** as the corner characters. See also **box** and **cbox**.

insch(c)
Inserts character **c** at the current (line, col) coordinates.

insertln ()
Inserts a line above the current line.

move(line, col)
Changes the current (line, col) coordinates of the window to (line, col).

overlay(win1, win2)
Overlays **win1** on **win2**. Windows need not be the same size.

overwrite(win1, win2)
Overwrites **win1** on **win2**. Windows need not be the same size.

printw(fmt, arg1, arg2, ...)
Performs a **printf** on the window starting at the current (line, col) coordinates.

refresh ()

Writes the contents of the specified window to the terminal.

standend ()
Stops putting characters onto **win** in standout mode.

standout ()
Starts putting characters onto **win** in standout mode.

Use the **refresh** routine to transfer the contents of the current window to the screen after all changes to the window are complete. The **refresh** routine does not rewrite any part of the window that has not changed since the last refresh call. To force the whole window to be rewritten, use the **touchwin** routine before the **refresh** routine. Also use **ecrfpn** to refresh a pane, and **ecrfpl** to refresh a panel.

9.4.3 Getting Input from the Terminal

Input is the complementary function to output. The screen package needs to know what is on the terminal at all times. Therefore, if a program echoes input characters, the terminal must be in a mode that passes characters immediately to the program, rather than waiting for a carriage return to send input to the program. The **getch** routine sets the terminal to the character input mode and then reads in the character.

Use the following routines for input from the terminal:

Routine **Description**

crmode ()
Sets the terminal to allow character by character input and not wait for a carriage return to send input to the process.

nocrmode ()
Sets the terminal to wait for a carrier return to send input to the process.

echo ()
Sets the terminal to **echo** characters.

noecho ()
Sets the terminal to not **echo** characters.

ecflin (pane, firstline, firstcol, numlines, numcols, pat, xc, buf, mask)
Gets input from the screen as long as the cursor is in a specified area (**field**) of the screen.

ecpnin (pane, boolt, xc)
Gets input from the screen in a specified pane, and scrolls the pane as needed to keep the cursor in the field.

extended ()
Controls international character support input processing.

getch ()
Gets a character from the terminal and (if necessary) echoes it on the window.

getstr(str)
Gets a string through the window and puts it in the location pointed to by **str**. The location must be large enough to hold the string. The string is terminated by **\n** (new-line).

keypad (boolf)
Controls the mapping of input from the keyboard. See "Controlling Input with the keypad, extended, and trackloc Routines" in topic 9.7.1 for more information.

raw ()
Sets the terminal to raw mode.

noraw ()
Resets the terminal from raw mode.

scanw(fmt, arg1, arg2, ...)
Performs a **scanf** through the window using **fmt**.

9.4.4 Controlling the Screen

Use the following library routines to control and manipulate the windows, panes, and panels on the screen.

Routine **Description**

delwin(win)
Deletes the window and frees the resources assigned to the window.

ecadpn (pane, win)
Adds the specified window to the list of windows that can be displayed in a pane, but does not display it.

ecaspn (pane, win)
Specifies a window to be displayed in the specified pane, but requires a **refresh** call to display it.

ecbpls (numlines, numcols, firstline, firstcol, title, divdim, border,pane)
Builds a panel structure.

ecbpns (numlines, numcols, ln, ld, divdim, ds, du, border, lh, lv)
Builds a pane structure.

ecdfpl (panel, boolf)
Creates **WINDOW** structures to define a panel.

ecdppn (pane, oldwin, newwin)
Removes the specified window from the list of windows that can be displayed in the pane.

ecdspl (panel)
Returns all structures associated with a panel to the storage pool, including structures for panes linked to the panel.

ecdvpl (panel)
Divides a panel into panes. All panes must be defined, and be linked to the panel.

ecrlpl (panel)
Returns structures associated with a panel to the storage pool, but not those that define the panel or the panes linked to the panel.

endwin ()
Restores the terminal to the state it was before **initscr** was called. Always use **endwin** before exiting.

gettmode ()
Gets the information about the terminal. This routine is called by **initscr**.

getyx(win, line, col)
Puts the current (**line**, **col**) coordinates of **win** in the variables **line** and **col**.

inch ()
Returns the character at the current (**line**, **col**) coordinates on

the specified window.

initscr ()
Initializes the screen routines. Call this routine before using any of the screen routines. Use the **endwin** before exiting the screen routines.

Routine Description

leaveok(win, boolf)
Sets the boolean flag **_leave** to the value specified by **boolf**. This flag indicates that the cursor should be positioned after the last change.

longname ()
Return the long (full) name of the terminal described by the terminfo entry.

mvcur(lastline, lastcol, newline, newcol)
Moves the terminal cursor from (**lastline, lastcol**) to (**newline, newcol**).

Note: Each window and the terminal have a cursor. The terminal cursor becomes the cursor on the active window or pane.

mvwin(win, line, col)
Moves the home position of the window **win** from its current starting coordinates to (**line**, **col**).

newview(orig_win, num_lines, num_cols)
Creates a new window that is **num_lines** lines and **num_cols** columns. The window is a viewport of the **orig_win** starting at the current (**line**, **col**) coordinates of **orig_win**.

newwin(lines, cols, begin_line, begin_col)
Creates a new window with **lines** lines and **cols** columns starting at position (**begin_line, begin_col**).

nl ()
Sets new-line mode so that the system starts changing return characters to linefeed characters.

nonl ()
Resets new-line mode so that the system does not change return characters. This setting helps the **refresh** routine perform optimization.

resetty (boolf)
Restores the tty characteristic flags to what **savetty** stored.

savetty ()
Saves the current tty characteristic flags.

scroll(win)
Scrolls the window upward one line.

scrollok(win, boolf)
Sets the scroll flag for the given window to the value specified by **boolf**. A value of FALSE (disable scrolling) is the default setting.

setterm(name)
Sets the terminal characteristics to be those of the terminal name **name**.

subwin(win, lines, cols, begin_line, begin_col)
Creates a new window with **lines** lines and **cols** columns starting at position (**begin_line, begin_col**) in the window **win**.

touchwin(win)
Forces the **refresh** routine to write all of the specified window, instead of just the parts that have changed.

trackloc (boolf)
Controls the tracking of the locator cursor.

Routine Description

tstp ()
When using the tty driver, this function saves the current tty state and then puts the process to sleep. When the process is started again, the process restores the tty state and then calls **wrefresh(curscr)** to redraw the screen. The **initscr** routine sets the signal SIGTSTP to trap to this routine.

unctrl(ch)
Returns a string which is a representation of **ch**. To use **unctrl**, put the statement:

```
#include <cur04.h>
```

in the program file.

vscroll(view_win, deltaline, deltacol)
Scrolls the viewport window (see **newview**) down **deltaline** lines and right **deltacol** columns. If the numbers are negative, the directions are up and left, respectively.

9.5 Routines for Panels and Panes

The **Extended curses** library contains routines to help create a screen appearance similar to that used for Usability Services. The following paragraphs describe the concept of the panel and pane interface, and list the routines for creating a panel and pane interface. Refer to *AIX Operating System Technical Reference* for detailed information about each routine.

Subtopics

9.5.1 Defining Panels and Panes

To define a panel, provide the following information about the panel:

The size of the panel as it appears on the displa

The location on the display of the upper left corner of the pane

Whether the panel is to have a border or no

How the panel is to be divided into panes

In addition, provide the following information for each pane within the panel:

The size of the presentation space associated with the pan

The relative size of the pane within the pane

Whether the pane is to have a borde

If and how the pane is to be further divided into smaller panes

To divide panels and panes into smaller panes, follow a few simple rules. These rules ensure that a program can access all areas on the panel or pane that it creates:

You can divide a panel or pane either horizontally (using a horizonta dividing line) or vertically (using a vertical dividing line)

Panes created by a horizontal division must be linked together fro top to bottom

Panes created by a vertical division must be linked together from lef to right

Panes that are divided again must be linked to the first pane of it sub-panes. The original pane in this case is not a part of the presented panel, but it is needed to define the structure of the panel

Panes created by a horizontal division have a fixed horizonta dimension that is the same as its parent pane

Panes created by a vertical division have a fixed vertical dimensio that is the same as its parent pane

Specify the variable dimension for a pane as being in one of thre categories:

Fixed For a **fixed** pane, specify the number of rows or columns, including any border, to assign to the pane.

Fractional For a **fractional** pane, specify the percentage of the available space to assign to the pane.

Floating For a **floating** pane, do not specify a size. The floating pane shares the available space equally with any other floating panes that the program creates.

The linkage of the panes forms a tree structure. The root of the tree is

a panel description. All other elements in the tree are pane descriptions.

9.5.2 Creating Panels and Panes

To create a panel that looks like the outline shown in Figure 9-3, perform the following steps (following the rules for dividing panels and panes) as shown in Figure 9-4.

1. Define panel **P** using the **ecbpls** routine with a link to pane **A**.

2. Divide the panel (**P**) with two horizontal splits into three panes. Use the **ecbpns** routine to define the three panes with the following links:

A	No links
B	Linked to **A** and **D**
C	Linked to **B** and **F**

3. Divide pane **B** with a single vertical split into two panes. Use the **ecbpns** routine to define the two panes with the following links:

D	No links
E	Linked to **D**

4. Divide pane **C** with two vertical splits into three panes. Use the **ecbpns** routine to define the three panes with the following links:

F	No links
G	Linked to **F**
H	Linked to **G**

Although the program must create panes **B** and **C** to get the smaller panes, those two panes do not appear as panes in the final display.

Figure 9-5 shows how the panel and pane descriptions for the final structure are linked. Horizontal lines show the links within a pane; vertical lines show links to the parent panel or pane.

The program in Figure 9-6 creates the panel shown in Figure 9-3.

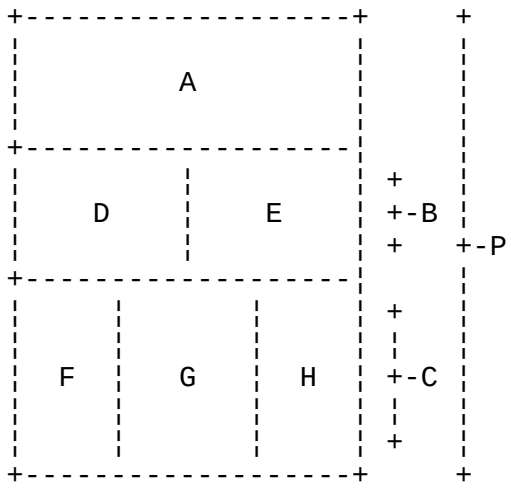


Figure 9-3. Example Panel Final Appearance

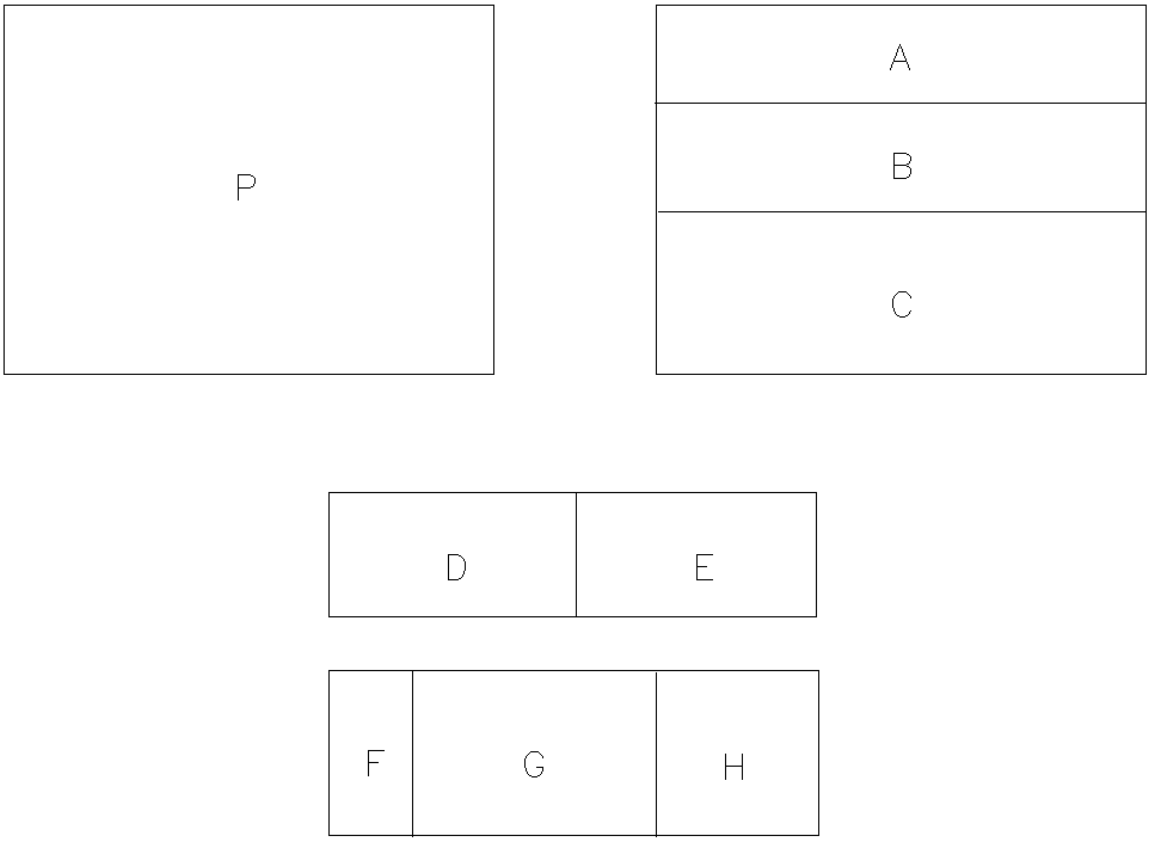


Figure 9-4. Creating Panes in the Panel

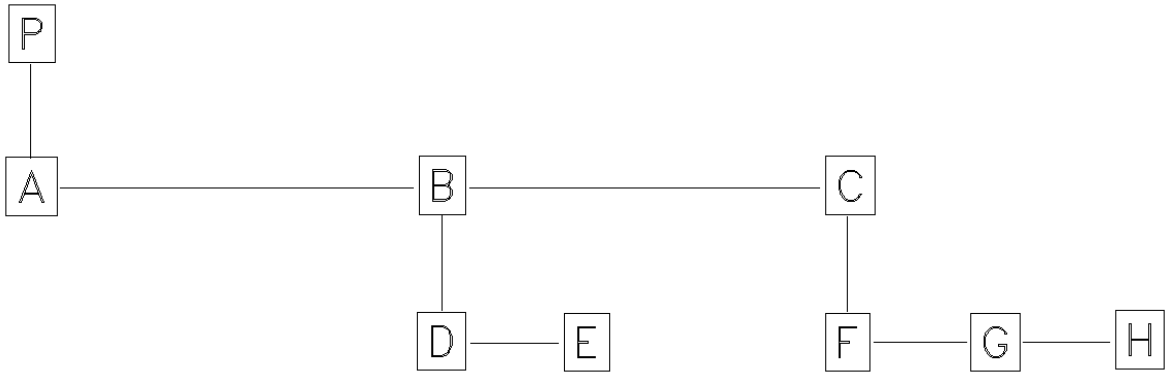


Figure 9-5. Links in the Panel and Pane Structure

```
  #include <cur01.h>
  #include <cur05.h>

  main()

  {
```

```
        PANE    *A, *B, *C, *D, *E, *F, *G, *H ;
        PANEL   *P ;

  initscr () ;

  A = ecbpns (24, 80, NULL, NULL, 0, 2500, Pdivszp, Pbordry, NULL, NULL);

  D = ecbpns (24, 80, NULL, NULL, 0, 0, Pdivszf, Pbordry, NULL, NULL);
  E = ecbpns (24, 80, D,    NULL, 0, 0, Pdivszf, Pbordry, NULL, NULL);

  B = ecbpns (24, 80, A, D, Pdivtyh, 3000, Pdivszp, Pbordry, NULL, NULL);

  F = ecbpns (24, 80, NULL, NULL, 0, 0,    Pdivszf, Pbordry, NULL, NULL);
  G = ecbpns (24, 80, F,    NULL, 0, 5000, Pdivszp, Pbordry, NULL, NULL);
  H = ecbpns (24, 80, G,    NULL, 0, 3000, Pdivszp, Pbordry, NULL, NULL);

  C = ecbpns (24, 80, B, F, Pdivtyh, 0, Pdivszf, Pbordry, NULL, NULL );

  P = ecbpls (24, 80, 0, 0, NULL, Pdivtyv, Pbordry, A );

  ecdvpl ( P );
  ecdfpl ( P, FALSE );
  ecshpl ( P );
  ecrfpl ( P );

  endwin();
  }   /* end main program */
```

Figure 9-6. Program to Create Example Panel

9.6 Display Attributes

Use the color and display characteristics defined in Figure 9-7. These names are external variables that define the attributes that a program can use on the current terminal. The values of these variables depend on the capabilities of the current terminal and the priorities that you assign to the attributes. Change the values of these variables with the **sel_attr** routine as explained in "Changing the Defined Attributes" in topic 9.6.1.

The characteristics that a program selects for the terminal are loaded into the attribute variable (of data type ATTR) associated with the data being displayed. Select as many of the attributes as needed, but those selected are packed into the attribute variable (of data type ATTR) in the following order:

1. REVERSE,
2. INVISIBLE,
3. F_WHITE,
4. F_RED,
5. F_BLUE,
6. F_GREEN,
7. F_BROWN,
8. F_MAGENTA,
9. F_CYAN,
10. F_BLACK,
11. B_BLACK,
12. B_RED,
13. B_BLUE,
14. B_GREEN,
15. B_BROWN,
16. B_MAGENTA,
17. B_CYAN,
18. B_WHITE,
19. BOLD,
20. UNDERSCORE, (only works with a terminal emulating an ega terminal)
21. BLINK,
22. DIM, (only works with a terminal emulating an ega terminal)
23. STANDOUT,
24. PROTECTED, (not supported on all terminal types)
25. FONT0, (not supported on all terminal types)
26. FONT1, (not supported on all terminal types)
27. FONT2, (not supported on all terminal types)
28. FONT3, (not supported on all terminal types)
29. FONT4, (not supported on all terminal types)
30. FONT5, (not supported on all terminal types)
31. FONT6, (not supported on all terminal types)
32. FONT7, (not supported on all terminal types)
33. NULL.

To change the order, see "Changing the Defined Attributes" in topic 9.6.1. Once the attribute variable (of data type ATTR) is full, the routines ignore the remaining lower priority attributes. If an attribute does not work with the current display, the routines ignore that attribute. Therefore, you can specify color attributes and still be able to use the program with a monochrome display. Figure 9-7 defines the external variable names that the routines use to set the display attributes.

Figure 9-7. Display Attributes

Name	**Attribute**

UNDERSCORE	Display characters with underline. (Only works with a terminal emulating an ega terminal).
REVERSE	Display characters in reverse video.
NORMAL	Display characters without highlighting (return to normal).
INVISIBLE	Do not display characters.
STANDOUT	Display characters in high intensity (can be used with other attribute colors). On many terminals, this is the same as **REVERSE**.
BOLD	Display characters in bold font (or high intensity on some terminals).
BLINK	Display blinking characters (can be used with other attribute colors).
DIM	Display characters in reduced intensity. (Only works with a terminal emulating an ega terminal).
PROTECTED	Protected display field. (This command is not supported on all terminal types.)
F_BLACK	Set foreground color to black.
F_BLUE	Set foreground color to blue.
F_GREEN	Set foreground color to green.
F_CYAN	Set foreground color to cyan.
F_RED	Set foreground color to red.
F_MAGENTA	Set foreground color to magenta.
F_BROWN	Set foreground color to brown.
F_WHITE	Set foreground color to white.
B_BLACK	Set background color to black.
B_BLUE	Set background color to blue.
B_GREEN	Set background color to green.
B_CYAN	Set background color to cyan.
B_RED	Set background color to red.
B_MAGENTA	Set background color to magenta.
B_BROWN	Set background color to brown.
B_WHITE	Set background color to white.
FONT0	Select defined character font 0.
FONT1	Select defined character font 1.
FONT2	Select defined character font 2.
FONT3	Select defined character font 3.
FONT4	Select defined character font 4.
FONT5	Select defined character font 5.
FONT6	Select defined character font 6.
FONT7	Select defined character font 7.

Subtopics
9.6.1 Changing the Defined Attributes
9.6.2 Changing Screen Attributes

9.6.1 Changing the Defined Attributes

To change the characteristics assigned to the external variables listed in Figure 9-7 in topic 9.6, use the **sel_attr** routine. This routine uses a set of defined constants contained in the header file **cur03.** To use this routine, put the following statement at the beginning of the program file:

```
#include <cur03.h>
```

The file **cur03.h** defines the following constants:

```
_dNORMAL
_dREVERSE
_dBOLD
_dBLINK
_dUNDERSCORE  (only works with a terminal emulating an ega terminal)
_dDIM (only works with a terminal emulating an ega terminal)
_dINVISIBLE
_dPROTECTED (not supported on all terminal types)
_dSTANDOUT
_dF_BLACK
_dF_RED
_dF_GREEN
_dF_BROWN
_dF_BLUE
_dF_MAGENTA
_dF_CYAN
_dF_WHITE
_dB_BLACK
_dB_RED
_dB_GREEN
_dB_BROWN
_dB_BLUE
_dB_MAGENTA
_dB_CYAN
_dB_WHITE
_dFONT0
_dFONT1
_dFONT2
_dFONT3
_dFONT4
_dFONT5
_dFONT6
_dFONT7
```

These constants are only valid when using the **sel_attr** routine. They cannot be used with any other routine.

9.6.2 Changing Screen Attributes

The code fragment in Figure 9-8 shows how to use these constants to change the default set of attributes. Note that the **_dDIM** and **_dUNDERSCORE** commands only work with terminals emulating an ega terminal.

```
#include <cur00.h>
#include <cur03.h>

int     attrs[] =
{
        _dBOLD, _dBLINK,
        _dF_WHITE, _dF_RED, _dF_BLUE, _dF_GREEN,
        _dF_BROWN, _dF_MAGENTA, _dF_CYAN, _dF_BLACK,
        _dB_BLACK, _dB_RED, _dB_BLUE, _dB_GREEN,
        _dB_BROWN, _dB_MAGENTA, _dB_CYAN, _dB_WHITE,
        _dREVERSE, _dINVISIBLE, _dDIM, _dUNDERSCORE,
        NULL
};

main( )
{

sel_attr(attrs);
initscr( );
if( REVERSE == NORMAL ) REVERSE = F_BLACK | B_WHITE;
if( INVISIBLE == NORMAL ) INVISIBLE = F_BLACK | B_BLACK;
if( DIM == NORMAL ) DIM = F_BLACK | BOLD;
if( UNDERSCORE == NORMAL ) UNDERSCORE = F_WHITE | B_RED;
STANDOUT = REVERSE;

        <rest of program>

endwin( );
}  /* end main */
```

Figure 9-8. Changing Screen Attributes Example Program

The routines define 16 bits of unique attribute information. Selecting foreground color, background color or font requires either 1, 2 or 3 bits depending upon the number of colors or fonts in the list: 1 bit for 2 or fewer, 2 bits for 3 or 4, and 3 bits for 5 to 8. Each character attribute takes 1 bit. However, the attribute names passed to **wcolorout** are variables, so that you can make combinations from the other attributes as shown in the last part of the previous example. If a requested attribute (that is not the terminal default) is equal to NORMAL, then it is either not supported by the terminal, or there is not enough space in the attribute variable (of data type ATTR) for its mask.

9.7 Using Other Features

Subtopics
9.7.1 Controlling Input with the keypad, extended, and trackloc Routines
9.7.2 Scrolling Windows
9.7.3 Improving Performance

9.7.1 Controlling Input with the keypad, extended, and trackloc Routines

The **keypad** routine allows a program to recognize control sequences in the input without searching the input or introducing device dependencies. If **keypad** is active, it scans all input data for control sequences. If it finds a control sequence, it returns the associated code to the program instead of the actual control sequence. The control codes are shown in Figure 9-9. These codes are defined in the file **cur02.h** with values greater than **0x100**.

To get international character support processing on input, **extended** must be active. If **extended** is turned off, shift codes and data codes input separately.

To get tracking of the locator cursor on the screen, **trackloc** must be active. If **trackloc** is turned off, the application has to handle the tracking of the locator cursor. For information about locator input, see *AIX Operating System Technical Reference*.

Figure 9-9. Control Codes

Name	**Description**
KEY_NOKEY	No keyboard data and no delay on
KEY_BREAK	Break
KEY_DOWN	Cursor down
KEY_UP	Cursor up
KEY_LEFT	Cursor left
KEY_RIGHT	Cursor right
KEY_HOME	Home - top left
KEY_BACKSPACE	Backspace
KEY_DL	Delete line
KEY_IL	Insert line
KEY_DC	Delete character
KEY_IC	Insert character mode start
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 forward
KEY_SR	Scroll backward (reverse)
KEY_NPAGE	Next page
KEY_PPAGE	Previous page
KEY_STAB	Set tab stop
KEY_CTAB	Clear tab stop
KEY_CATAB	Clear all tab stops
KEY_ENTER	Enter key
KEY_SRESET	Soft reset key
KEY_RESET	Hard reset key
KEY_PRINT	Print or copy
KEY_LL	Lower left (last line)
KEY_A1	Pad upper left
KEY_A3	Pad upper right
KEY_B2	Pad center
KEY_C1	Pad lower left
KEY_C3	Pad lower right
KEY_DO	DO key
KEY_QUIT	QUIT key
KEY_CMD	Command key
KEY_PCMD	Previous command key
KEY_NPN	Next pane key

KEY_PPN	Previous pane key
KEY_CPN	Command pane key
KEY_END	End key
KEY_HLP	Help key
KEY_SEL	Select key
KEY_SCR	Scroll right key
KEY_SCL	Scroll left key
KEY_TAB	Tab key
KEY_BTAB	Back tab key
KEY_NEWL	New-line key
KEY_F0	Function key - 128 values
KEY_F(n)	Not used
KEY_ESC1	Added to the ending character code for ESC sequences in the form ESC c with c in the range 0x30 - 0x7f. The value sent is in the range 0x200 to 0x24f.
KEY_ESC2	Added to the ending character code for ESC sequences in the form ESC [s c with c in the range 0x40 - 0x7f. The value sent is in the range 0x250 to 0x28f.

To use the control sequences in a program, first use a call to the **keypad** routine:

```
keypad(TRUE);
```

9.7.2 Scrolling Windows

If a program defines a window with **scrollok**, scrolling is allowed for that window. When the program writes a character to the lower right corner of the window (or adds **\n** to the last row of the window), the routines automatically call the **scroll** routine.

If a program does not define the window with **scrollok**, scrolling is not allowed for that window. When the program writes a character to the lower right corner of the window (or adds **\n** to the last row of the window), the routines reset the current **col** coordinate to zero (beginning of line) and the window does not scroll.

If a window includes the lower right corner of the terminal screen, the flag byte bit, **_SCROLLWIN**, in the WINDOW structure for that window is set. This bit indicates that if a character is written to the lower right corner of the window, the terminal (*not* the routines) inserts a blank line at the bottom of the screen (scrolls) to make room for more information. To avoid this hardware scroll, the routines operate differently for a window that includes the lower right corner of the terminal screen and is not defined with **scrollok**. When a program writes a character to the lower right corner of such a window (and screen), the character is added to the data array for the window but the character is not written to the window when the window is refreshed.

To move a window to the lower right corner, use the **mvwin** routine. The **_SCROLLWIN** flag bit for that window is not automatically set. However, the **wrefresh** routine handles that window as if the **_SCROLLWIN** flag bit were set.

9.7.3 Improving Performance

To speed up output, create an output buffer using statements similar to the following program fragment:

```
#include  <stdio.h>

char      obuf[BUFSIZ];

main( )
{
   setbuf (stdout, obuf);
      .
      .
      .
   /* rest of program */
}
```

9.8 Example Program

Figure 9-10 shows the use of some of the routines to create a series of displays on the screen. Compile and run the program to see the effects of the Extended curses functions. To compile the example program, use the following command:

```
cc twinkle.c -lcur -lcurses
```

```
#include        <cur00.h>
#include <signal.h>

#define NCOLS 80
#define NLINES 24
#define MAXPATTERNS 11

struct locs
{
   char  y, x;
};

typedef struct locs  LOCS;

LOCS  layout[ NCOLS * NLINES ];  /* current board layout */


int   pattern,                   /* current pattern number */
      numstars;                  /* numbers of stars in ptern */

main()
{
   char *getenv();
   int die();

   srand( getpid() );            /* initialize random sequence */
   initscr();
   signal( SIGINT, die );
   noecho();
   leaveok( stdscr, TRUE );
   scrollok( stdscr, FALSE );

   for( ;; )
   {
      makeboard();               /* make the board setup */
      puton( '*' );              /* put on '*'s */
      system( "sleep 2" );
      erase();
      refresh();
   }
}


die()
{
   signal( SIGINT, SIG_IGN );
   mvcur( LINES/2, COLS/2, 0, 0 );
```

```
   wclear( curscr );
   wrefresh( curscr );
   endwin();
   exit(0);
}
makeboard()
{
   reg int  y, x;
   reg LOCS  *lp;
   pattern = rand() % MAXPATTERNS;
   lp = layout;
   for( y = 0; y < NLINES; y++ )
   {
      for( x = 0; x < NCOLS; x++ )
      {
         if( ison( y, x ))
         {
            lp -> y = y;
            lp++ -> x = x;
         }
      }
   }
   numstars = lp - layout;
}
ison( y, x )

reg int y, x;

{
   switch( pattern )
   {
      /*
      ** Alternating lines:
      */
      case 0:
         return !( y & 01 );
      /*
      ** Box:
      */


      case 1:
         if( y < 3 || y >= NLINES - 3 )
            return TRUE;
         return( x < 4 || x >= NCOLS - 4 );
      /*
      ** Cross:
      */

      case 2:
         return( ( x + y ) & 01 );
      /*
      ** Bar across center:
      */

      case 3:
         return( y >= 9 && y <= 15 );
      /*
      ** Alternating columns:
      */
```

```
      case 4:
         return !( x & 02 );
      /*
      ** Bar down center:
      */

      case 5:
         return( x >= 36 && x <= 44 );
      /*
      ** Bar across and down center:
      */

      case 6:
         return( ( y >= 9 && y <= 15 ) || ( x >= 37 && x <= 43 ));
      /*
      ** Bar across and down center, in a box:
      */


      case 7:
         if( y < 3 || y >= NLINES - 3 )
            return TRUE;
         if( x < 4 || x >= NCOLS - 4 )
            return TRUE;
         return( ( y >= 10 && y <= 14 ) || ( x >= 36 && x <= 44 ));
      /*
      ** Asterisk:
      */

      case 8:
         if( abs( x - y ) <= 2 || abs( NLINES - ( x + y )) <= 2 )
            return TRUE;
         if( abs( ( NLINES/2 ) - x ) <= 2 )
            return TRUE;
         return( abs( ( NLINES/2 ) - y ) <= 1 && x <= NLINES );
      /*
      **  Ellipse:
      */

      case 9:
         return
         (
            (
               (( float ) (( x-40 ) * ( x-40 )) ) / 1521 +
               (( float ) (( y-12 ) * ( y-12 )) ) / 121
            ) <= 1
         );
      /* Circle: */

      case 10:
         return
         (
            (
               (( float ) (( x-28 ) * ( x-28 )) ) / 729 +
               (( float ) (( y-12 ) * ( y-12 )) ) / 121
            ) <= 1
         );
   }    /* end of switch( pattern ) */
}       /* not reached */
```

```
  puton(ch)
  reg char ch;
  {
     reg LOCS  *lp;
     reg LOCS  *end;
     LOCS      temp;
     reg  int     r;


     end = &layout[ numstars ];
     for( lp = layout; lp < end; lp++ )
     {
        r = rand() % numstars;
        temp = *lp;
        *lp = layout[ r ];
        layout[ r ] = temp;
     }
     for( lp = layout; lp < end; lp++ )
     {
        mvaddch( lp -> y, lp -> x, (NLSCHAR) ch );
        refresh();
     }
  }  /* end of twinkle */
```

Figure 9-10. Example of Extended curses Program

10.0 Chapter 10. Writing Messages and Help

Subtopics

10.1 CONTENTS

10.2 About This Chapter

This chapter explains how to use the operating system services to provide both messages and help from a program. It describes the message and help text files, how to make them, and how to incorporate them into a program. The chapter also describes the format of messages.

10.3 Messages

A ***message*** is information that the program generates to inform the person using the program, or **/dev/console** of conditions in the program. If the conditions require steps to recover, the message provides those steps. A program can generate two types of messages:

Immediate message
The message appears on the screen associated with the program. The message is usually in response to something that the person using the program did.

Queued message
The message appears in the message queue file **/qmsg** and can only be seen by listing or editing the message queue file. When a message enters the queue file, a beep tone notifies **/dev/console**. Programs that operate directly with the user usually do not produce queued messages. Background processes, such as daemons, produce queued messages.

The operating system provides a set of routines, called ***message services*** to help create, update and display messages from a program. The routines are in the library file **/usr/lib/librts.a**. The services for generating messages include:

A standard message format that matches the format of the operatin system messages

A file containing a template to use to create message

Two routines that help to generate either immediate or queued message from a program

Header files to simplify declarations needed to use message service

Variable field symbols in the messages. When message service displays the message, it replaces these symbols with values that you specify.

Subtopics
10.3.1 Message Format

10.3.1 Message Format

Each of the two types of messages has a different format. When displayed, queued messages are in the following format:

```
MM/DD  HH:mm  y       z  pgm-nnn
This is a sample queued message.
```

Immediate messages have the following format.

```
pgm-nnn This is a sample immediate message:
                a. Short line.
                b. This is a long line.  Note that
                   the message is presented
                   as you format it. It is not
                   reformatted before being displayed.
        Time = HH:mm.  Severity = y.  Error Number = z.
```

The symbols have the following meaning. See "Using Routines to Display Messages" in topic 10.5.2 for a description of the library routines for generating messages.

Figure 10-1. Message Fields

Field	**Description**
pgm	A 3-character program identifier that is unique to the program. To ensure that these characters do not conflict with numbers already assigned to system programs, choose a number that is larger than 500. You can also use alphabetic characters for the program identifier. Figure 10-2 shows some of the identifiers that the system programs use. To match the style of the system messages, choose three digits for the program identifier. Message services displays this identifier to help the operator know what program generated the message. Message services do not use the identifier as an index into the messages.
nnn	A 3-digit sequence number for the message within the set of messages for the program identifier specified by *pgm*. This number allows looking up a description of the message in a book. Message services do not use the number.
text	The words that explain the condition associated with the message.
MM/DD	The month and day that the message was generated.
HH:mm	Time (24 hour format) that the message was issued. When a program generates an immediate message with the **msgimed** function and uses the **msgfltim** flag, message services supplies the time. Message services automatically provides the time for queued messages.
y	Severity code is the severity code specified when using the message services routines to write the message. If you do not specify a severity code, this field does not appear.

z Error number is the error code specified when using the message services routines to write the message. If you do not specify an error code, this field does not appear.

For example, if the operator makes a mistake when entering the date, the program could generate the following message:

```
345-007 The system cannot recognize the date that you
        entered.  Please enter the date again.
```

The optional information (time, severity or error code) does not appear in this message because the program did not specify that they be displayed when it called for the message. The book for this program should include an entry for error number **345-007** that contains information about the correct date format, or other information to help correct this problem.

Figure 10-2. System Identifiers

ID	**System Program**
000	Common
001- 007	Not Available
008	Operating System Extension Install
009 - 012	Not Available
013	Operating System Install and Maintenance
014 - 020	Not Available
021	Operating System Trace Points
022 - 033	Not Available
040 - 041	Operating System Configuration
042 - 044	Not Available
046	Install and Update Services
047	Not Available
048	Install and Update Services
049 - 060	Not Available
061	Dialog Manager
062	Application Developers Toolkit
063 - 068	Not Available
069	Activity Manager
070 - 074	Not Available
075	Hardware Access Support
076	Base LAN Install
077	Pascal Compiler
078	Not Available
079	Device Drivers - ports
080 - 081	Not Available
082	Print I/O Services
083 - 089	Not Available
090	Message Services
091	Tools Application
092 - 094	Not Available
095	Operating System
096	Dialog Definition Statements
097	Files Application
098 - 102	Not Available
103	Operating System
104 - 105	Not Available
106	Not Available
107	Not Available
108	**dumpfmt** Command
109	Error Log

```
110                 Trace
111 - 499           Not Available
500 +               Available for new programs
```

10.4 Building a Message Table

The services of the operating system can help to build a table of messages that is separate from the source code of the program. To build the table of messages, first get a file containing the standard message format from the file system. Then add messages to that file, compile the table of messages (using **cc**), and link the messages with the compiled program (object modules) and the messages library **/lib/librts.a**. This method keeps the messages in memory when the program is in memory. Do not use this method for long text, such as help (see "Help" in topic 10.7).

Having a separate table of messages makes it easier to change messages, add messages, and translate the messages to another language.

Perform the following steps to build a message table and incorporate it in the program. Refer to the following paragraphs for additional explanation for some of the steps:

1. Copy the example message table file **msg07.h** into the current directory:

   ```
   cp /usr/include/msg07.h .
   ```

2. Rename the example file to the name of the message source file. Use a **.c** file extension.

   ```
   mv msg07.h mymsgs.c
   ```

3. Replace the name of the table, *tablename* in the example file, with the external name of the table.

4. Use an editor to add the message definitions and text to the message table source file.

5. Compile the message table and program source files using the **cc** command.

   ```
   cc program-files.c mymsgs.c -o myname
   ```

   In this command, **program-files.c** can be any number of C language source files each with a **.c** extension. The resulting executable program is in the file **myname**.

Subtopics

10.4.1 Copying the Standard Format File

In the directory **/usr/include** is a file called **msg07.h**. This file is an ASCII file that contains the framework for building a message table. Figure 10-3 shows the major parts of this file.

```
                      /*******************/
   #define TABLE_NAME /***/ tablename /***/
                      /*******************/

   #include <msg08.h>
                       /* structure declarations */
  /*
  **          ** MESSAGE DEFINITIONS **
  */
   static msg__msg msg_defs[] = {

   0, "345", "007",
   "The system cannot recognize the date that you \n\
    entered.  Please enter the date again.",
                                  /* message 001 */
   } ;
  /*
  **          ** TEXT INSERT DEFINITIONS **
  */
   static msg__ins ins_defs[] = {

   "month",                       /* insert 001 */
   "day",                         /* insert 002 */
   "year",                        /* insert 003 */
   } ;

   #include <msg09.h>  /* pointers table */
```

Figure 10-3. Content of Message Standard Format File

10.4.2 Naming the Message Table

The name of the table is the name assigned to it in the first line of the table file. The following line defines a table name of **xyztab1**.

```
#define TABLE_NAME xyztab1
```

For consistency with system table conventions, use the following guidelines when naming the table:

The first three characters should be the program identifier **xyz** in the example).

The next three characters should be the letters **tab** to indicate a table.

The last character should be an identifier to set this table apar from other tables in the program (**1** in the example).

10.4.3 Adding Message Definitions

The ***message definition*** is the entry in the message table that describes the message. For example, in the previous standard format file, the entry:

```
0,"345","007",
"The system cannot recognize the date that you \n\
entered.  Please enter the date again.",
```

is a message definition.

The message definition has the following parts:

0 The first number should be the index number for the help for the message. In the example, the number **0** indicates that help is not available for the message. A positive number in this position is the index number for help. A negative number in this position is the index number for help contained in the common help file. This number can only be used by the dialog manager.

program identifier
The second field (**345** in the example) is a 3-character field that identifies the program. It must not conflict with identifiers already used by the system programs. Figure 10-2 in topic 10.3.1 shows the identifiers that the system programs use. The identifier is enclosed in quotes.

message number
The third field (**007** in the example) is a 3-digit field that indicates the number of the message within all messages for the program identifier. This number helps to find the message in the documentation. Message services does not use it.

message text
Messages can be any length, but must include \n (new-line) characters as appropriate to keep the final output line length, including expanded variable fields (see "Using Variable Fields in Message Text" in topic 10.6), to no longer than 71 characters. Message text is enclosed in quotes.

, A comma that is not enclosed in a set of quotes marks the parts and the end of a message definition.

}; A right brace followed by a semicolon that is not enclosed in a set of quotes indicates the end of all message definitions.

10.4.4 Message Index

The message index is the position of the message definition within all message definitions in the table. The first definition is number 1; the second definition is number 2. The index does not depend on any number in the table file, only on the order of the message definitions in the file. Therefore, to delete a message definition, replace it with a null definition. If you remove a definition without providing a place-holding null definition (or a new definition in that position), the index to all message definitions that occur later in the file will change.

10.4.5 Adding Text Insert Definitions

The text insert definition is the entry in the message table that describes the fixed text strings to insert into messages in place of the symbols **@T1**, **@T2** or **@T3**. If one or more of these symbols is in the message definition, the program can select any text insert string from the message table to replace that symbol. The method for doing that is described in "Using Variable Fields in Message Text" in topic 10.6.

The requirements of the text insert definition section of the message table are that it:

Follows the message definitions sectio

Begins with the statement

```
static msg__ins ins_defs[] = {
```

Contains a series of text strings that are

- Enclosed in quotes
- Separated by commas

Ends with a **};** (right brace and semicolon).

The example message table in Figure 10-3 in topic 10.4.1 shows an example of the text insert definition section. That example contains the following text strings:

```
"month",
"day",
"year",
};
```

10.5 Using Messages in a Program

To use the operating system message services to display messages from a message table, the program should:

1. Include the correct header files in the program.

2. Pass needed values to message services by setting system external variables (see "Using Variable Fields in Message Text" in topic 10.6).

3. Use the message services routines to generate the messages.

Subtopics
10.5.1 Including Header Files
10.5.2 Using Routines to Display Messages

10.5.1 Including Header Files

Message services uses the header files shown in Figure 10-4. Include these files in the program when using message services. These header files are in the directory **/usr/include**. Display these files (using the **pg** command) to see the exact content.

Figure 10-4. Header Files

File Name	Function
msg00.h	Contains **#include** statements for the following main header files to make including those files easier: msg01.h msg02.h msg03.h msg04.h msg05.h msg06.h msg08.h
msg01.h	Defines the bits of the **flags** argument to message routines.
msg02.h	Defines severity codes displayed in messages.
msg03.h	Defines origin codes that indicate where the error was detected.
msg04.h	Defines error return codes.
msg05.h	Defines a structure for the **msgrtrv** function when a zero value is specified for the **nbyte** argument.
msg06.h	Defines the external variables that pass values to message services.
msg08.h	Defines structures to be used when compiling a message table.

10.5.2 Using Routines to Display Messages

After including the needed header files and setting any needed values in the message services external variables, use one of the following routines to display the message. The message can be either immediate or queued.

Subtopics
10.5.2.1 Generating an Immediate Message
10.5.2.2 Generating a Queued Message

10.5.2.1 Generating an Immediate Message

The **msgimed** routine performs the following functions:

1. Gets message text from the message table.

2. Expands standard symbols in the message text.

3. Outputs the message to either **stderr** or to a specified file.

Refer to *AIX Operating System Technical Reference* for information about the format and syntax of this routine.

10.5.2.2 Generating a Queued Message

The **msgqued** routine performs the following functions:

1. Gets message text from the message table.

2. Expands standard symbols in the message text.

3. Outputs the message to the queued message file **/qmsg**.

Note: Queued messages are sent to **/qmsg** and generate a beep tone at the system console. They are not directed at the person using the program.

Refer to *AIX Operating System Technical Reference* for information about the format and syntax of this routine.

10.6 Using Variable Fields in Message Text

A variable field is a standard symbol in the text of the message that is replaced with a value when message services displays the message. The value can be either a static (unchanging) text string or a new value each time, depending on the type of variable field used. The program passes values to message services by setting external variables. Use the following standard symbols in message definitions:

Figure 10-5. Standard Symbols

Symbol	Definition
@I1	Message services replaces this symbol with the character form of the integer value in the external variable **msgvi1**.
@I2	Message services replaces this symbol with the character form of the integer value in the external variable **msgvi2**.
@L1	Message services replaces this symbol with the character form of the long integer value in the external variable **msgvl1**.
@L2	Message services replaces this symbol with the character form of the long integer value in the external variable **msgvl2**.
@C1	Message services replaces this symbol with the null-terminated character string pointed to by the external variable ***msgvc1**.
@C2	Message services replaces this symbol with the null-terminated character string pointed to by the external variable ***msgvc2**.
@C3	Message services replaces this symbol with the null-terminated character string pointed to by the external variable ***msgvc3**.
@T1	Message services replaces this symbol with the text insert (from the text insert definitions section of the message table) selected by using the external variable **msgvt1** as an index value.
@T2	Message services replaces this symbol with the text insert (from the text insert definitions section of the message table) selected by using the external variable **msgvt2** as an index value.
@T3	Message services replaces this symbol with the text insert (from the text insert definitions section of the message table) selected by using the external variable **msgvt3** as an index value.

Subtopics

10.6.1 Example of the Integer Symbol

If a message definition contains the message:

  "Your value of @I1 is out of range."

To display the value that the operator entered, assign that value to the external variable **msgvi1** in the program before calling for the message to be displayed.

Figure 10-6 shows an outline of a program to display that message. In this program, if index 3 of **xyztab1** contains the message definition of the previous example, and the value of **val** is **14**, message services displays the following message:

  Your value of 14 is out of range.

```
  /* Define integer to hold operator input */
  int val;
  /* Define integer for return code */
  int i;
  /* Include required files to use message services */
  #include <msg00.h>

  {
     extern msg__table xyztab1;

  /*
  ** Code that handles input and determines
  ** that the value is out of range.
  */

  /*
  ** Assign bad value to external variable, and
  ** call the routine to display an immediate message
  ** to standard error, using message index 3
  ** and message table, xyztab1.
  */
  msgvi1 = val;
  i = msgimed(MSGFLTAB,&xyztab1,3);
  /*
  ** Rest of program
  */
  }
```

Figure 10-6. Example of Integer Symbol Programming

10.6.2 Example of the Long Integer Symbol

Inserting a long integer value into a message is the same as inserting an integer value, except it uses the external variable **msgvl1** to assign a value to the symbol **@L1**, and the variable assigned to the **msgvl1** must be of type **long**. With those exceptions, use the example for integer values as a framework to use a long integer value.

10.6.3 Example of the Character String Symbol

If the message definition contains the message:

```
"The day that you entered, @C1, is not\n\
 a correct day of the week.  Please try again."
```

To display the string that the operator entered, assign a pointer to that string to the external variable **msgvc1** in the program. Then call for the message to be displayed. The string must end with a **\0** (null character).

Figure 10-7 shows the outline of a program to display that message. In this program, if index 4 of **xyztab1** contains the message definition of the previous example and the string entered is **munday**, message services displays the following message:

```
The day that you entered, munday, is not
a correct day of the week.  Please try again.
```

```
/* Define pointer to operator input. */
char *inp;
/* define integer for return code. */
int i;
/* Include required files to use message services. */
#include <msg00.h>

{
   extern msg__table xyztab1;

/*
** Code that handles input and determines
** that the string entered is not correct.
** Code sets inp to point to the null
** terminated string received from the user.
*/

/*
** Assign pointer inp to external variable, and
** call the routine to display an immediate message
** to standard error, using message index 4
** and message table, xyztab1.
*/
 msgvc1 = inp;
 i = msgimed(MSGFLTAB,&xyztab1,4);

/*
 rest of program
*/
}
```

Figure 10-7. Example of Character String Symbol Programming

10.6.4 Example of Text Insert Symbol

If the message definition contains the message:

```
"The @T1 that you entered is not\n\
 a correct @T1.  Please try again."
```

Use this message to indicate errors in many areas of the program. First create a ***text insert definition*** section in the message table. See "Adding Text Insert Definitions" in topic 10.4.5 for the format of a text insert definition section. If the text insert definition section contains the following few entries at the beginning of the table:

```
"day",          /* index 1 */
"week",         /* index 2 */
"month",        /* index 3 */
"year"          /* index 4 */
```

you can insert any of the strings **day**, **week**, **month** or **year** in place of the symbol **@T1** in the message by setting the external variable **msgvt1** to the index value (1, 2, 3 or 4) that selects the string to insert. When displaying the message, message services looks up the text string in the message table and inserts it in place of the symbol **@T1**.

Figure 10-8 shows the outline of a program that displays this message. In this example, if index 3 of **xyztab1** contains the message definition of the previous example and the value of **i_text** is 3, message services displays the following message:

```
The month that you entered is not
a correct month.  Please try again.
```

```
/* Define integer for return code. */
int i;
/* Define integer to select insert text. */
int i_text;
/* Include required files to use message services. */
#include <msg00.h>

{
   extern msg__table xyztab1;

/*
** Code that handles input and determines
** that a field entered is not correct.
** Code sets i_text to the index value to
** select the name of the field in error.
*/

/*
** Assign i_text to external variable, and
** call the routine to display an immediate message
** to standard error, using message index 3
** and message table, xyztab1.
*/
 msgvt1 = i_text;
 i = msgimed(MSGFLTAB,&xyztab1,3);
```

```
  /*
   rest of program
  */
  }
```

Figure 10-8. Example of Text Insert Symbol Programming

10.7 Help

Help is text that explains difficult concepts, quick procedure steps, or other information to make it easier to use the program. You determine when to display the help from the program, and what information to include in the help. Message services provides a way to incorporate help into the program while lowering memory usage.

You can use help in the program with all of the variable symbols used with messages, including text inserts. Define text inserts in the same file that defines the help text.

Because help usually contains a lot of text, help is not compiled into the program like messages are. Instead, help is in a specially formatted help file that is not kept in memory unless it is being used. Overlaying the help file in this manner helps lower memory requirements for the program.

Note: The method for storing and displaying help described in this part of the book can also be used for messages that are not kept in memory with the program. You can put both help definitions and message definitions in the same file.

Message services provides a routine to help display help from the program. The routine is in the library file **/usr/lib/librts.a**. When using message services, link this library file with the program. In addition, two programs **gettext** and **puttext** help to change or create information in the help files. The services include:

A routine that displays help from the program

Header files to simplify declarations needed to use message services

Variable field symbols in the help that message services replaces wit values that you specify when it displays the help.

Subtopics

10.7.1 Help Format

Choose any format for the help text that fits the program. You can use the format described for messages (see "Message Format" in topic 10.3.1), you can change that format, or you can define a different format. However, follow the format defined for the help file.

10.7.2 File Path Name

If you do not change the default path, the file that contains the help text must have the path name of:

/usr/lib/msg/**program**_EN.m

where the file name parts are:

program
: A program identifier that is unique.

_EN.m
: The ending sequence that message services requires of all help (and message) files that are not linked with the program.

10.7.3 Changing the File Path Name

You can specify an alternative directory to contain the help and message file for the program. Use the rules for naming the help file as described in "File Path Name" in topic 10.7.2. To specify the alternative directory, assign a pointer to the new path name prefix to the external variable **msgpath** in your program. For example:

```
msgpath = "/u/myprog/";
```

This statement tells message services to look in directory **/u/myprog** for the new help file. Message services looks in the default directory only if it cannot find the file in the directory you specify with the **msgpath** variable. If **msgpath** contains a null value, message services looks only in the default directory. To specify the current directory as the directory that contains the help file, set the **msgpath** variable to a pointer to a null string.

The **msgpath** variable is declared in **msg06.h**.

10.7.4 Changing the File Path Name for Debugging

You can also specify an alternative path name to use for the help file when testing help definitions. An alternative path name allows you to test new or changed help definitions without affecting the existing help file that is installed on the system. You can also use the alternative path name to test a new help file before installing it on the system. Use the rules for naming the help file as described in "File Path Name" in topic 10.7.2.

To specify the alternative directory, assign the new path name prefix to the environment variable **MSGPATH** from the command line. For example:

```
MSGPATH=/u/mytest/
```

assigns the path name prefix **/u/mytest/** to the **MSGPATH** variable. Then, export the variable with the **export** command on the command line:

```
export MSGPATH
```

This operation tells message services to look in directory **/u/mytest** for the help file. Message services looks in the default directory only if it cannot find the file in the directory specified with the **MSGPATH** variable.

10.8 Building a Help File

Message services can help to build a file of help text that is separate from the source code of the program. To build the help file, first get a file containing the standard help format from the file system. Then add help definitions to that file, and, using a program from message services, format that file for use as a help file. This method puts the help text in a file that message services can read.

Having a separate help file makes it easier to change the help text, add help text, and translate the text to another language.

Perform the following steps to build a help file and format it for use by message services. Descriptions of the commands **gettext** and **puttext** are in *AIX Operating System Commands Reference*. Refer to the following for additional explanation.

1. Use the **gettext** command to create a file containing the input format for a help file. This file contains formats to fill in for including messages and text inserts, as well as help text, in the help file. For example:

   ```
   gettext myhelps
   ```

   creates a file **myhelps** that contains the framework for a help file.

2. Add help text to the proper places in the file.

3. Use the **puttext** command to format the help file. For example:

   ```
   puttext myhelps myprog_EN.m
   ```

   uses the help text in the file **myhelps** to create the help file **myprog_EN.m**. See "File Path Name" in topic 10.7.2 for information about naming the help file.

Subtopics
10.8.1 Content of the Help Text File

10.8.1 Content of the Help Text File

When the **gettext** command gets a help file format to fill in, the file that the command creates contains entries like those in Figure 10-9.

```
-------------------------------------------------------------------------

  COMPONENT ID = prgxxx

  **(((Start message)))************************************
  INDEX#:
  COMPSRC:
  MSGSRC:
  DCOMPID:
  DMSGID:
  STATUS:
  HELP#:
  TEXT:

  **(((Start insert)))*************************************
  INDEX#:
  COMPSRC:
  MSGSRC:
  DCOMPID:
  DMSGID:
  STATUS:
  TEXT:

  **(((Start help)))***************************************
  INDEX#:
  COMPSRC:
  MSGSRC:
  DCOMPID:
  DMSGID:
  STATUS:
  TITLE:
  TEXT:


-------------------------------------------------------------------------
```

Figure 10-9. Content of Help Text Format File

The fields shown in the figure have the following meaning:

COMPONENT ID = prgxxx
Replace **prgxxx** with a 6-character identifier to indicate which program uses this help file. Do not use the characters: *** ? [ ]** and blank.

****(((Start** *typename***)))*********************
This line is a delimiter that starts each new message and must be in the format indicated. Replace *typename* with either **message**, **insert** or **help**, as appropriate.

INDEX#:
A 3-digit field that indicates the number of the message within each type (message, insert or help). Numbers start with 001 at the beginning of the definitions for each type. Message services uses this index number to locate the text to be displayed.

COMPSRC:
Component source - Defines where the text definition for this message is located. Enter **======** for *in this file*.

MSGSRC:
Message source - A 3-digit field that defines the index number to use to display text for this message. Enter either **===** to indicate the current **INDEX#**, or a number to get text from a different message in this file.

DCOMPID:
Component ID - A 3-character field that identifies the program. It must not conflict with identifiers already used by the system programs.

DMSGID:
Message ID - A 3-digit field that indicates the number of the message within all messages for the program identifier. This number helps to find the message in the documentation. Message services does not use it.

STATUS:
Status of the message - Enter **null** to indicate the message is not used. Enter **current** to indicate the message is an active message.

HELP#:
A 3-digit index number to locate the help text for a message. This field is not required. Access this field using the **msgrtrv** call.

TITLE:
A character field of up to 79 characters that can provide a title for help text displayed on the screen. You can access this field only with the **msgrtrv** call, not with the **msghelp** call.

TEXT:
Enter the text for the message, insert or help, formatted as it appears on the screen.

Help text, like immediate messages, can be as long as needed to cover the information. Do not include any message ID for help text unless it refers to the book supplied with the program. You can also use variable fields in the help text. Refer to "Using Variable Fields in Message Text" in topic 10.6 for information about using variable fields.

10.9 Using Help in a Program

To use the operating system message services to display help from the help file, the program must:

1. Include the correct header files in the program.

2. Pass needed values to the message services by setting system external variables (see "Using Variable Fields in Message Text" in topic 10.6).

3. Use the message services routines to display the help text.

Subtopics
10.9.1 Including Header Files
10.9.2 Using Routines to Display Help

10.9.1 Including Header Files

The operating system message services uses header files. Include these files in the program when using message services. These header files are described in "Including Header Files" in topic 10.5.1.

10.9.2 Using Routines to Display Help

After including the needed header files and setting any needed values in the message services external variables, use the following routines to generate the help.

Subtopics
10.9.2.1 Displaying a Help
10.9.2.2 Putting Help in a Buffer

10.9.2.1 Displaying a Help

The **msghelp** routine performs the following functions:

1. Gets help text from the help file.

2. Expands standard symbols in the help text.

3. Outputs the help to either **stderr** or to a specified file.

Refer to *AIX Operating System Technical Reference* for information about the format and syntax of this routine.

10.9.2.2 Putting Help in a Buffer

The **msgrtrv** routine performs the following functions:

1. Gets help text from the help file.

2. Expands standard symbols in the help text.

3. Outputs the help to a specified buffer.

Refer to *AIX Operating System Technical Reference* for information about the format and syntax of this routine.

11.0 Chapter 11. Monitoring Program Activities

Subtopics

11.1 CONTENTS

11.2 About This Chapter

This chapter contains information on monitoring software. It includes information on the following:

Trace facilities - The trace facilities allow you to log and forma trace data. This includes a functional definition of the trace components and information on using the trace commands and subroutines.

Error log facilities - These facilities are similar to the trac facilities except that you log error data rather than trace data.

Dump facilities - The dump facilities are used to analyze data store that was stored in memory at the time of a system failure.

11.3 Monitoring Program Activities

The system provides several software components that enable you to monitor program activities. These components can be grouped into three basic categories:

Trac

You use the trace facilities to monitor system performance or to aid in debugging programs. The system programs have several event classes, each of which can be turned on or off depending on your needs. For each event class, several trace points have been defined in various system software components. When a component containing a trace point is processed, the trace point generates a trace entry.

You can also generate trace entries from your own programs using the three subroutines provided with the system. These subroutines allow you to generate trace entries from applications or AIX Operating System kernel components. All trace entries are stored in a **trace log** file. The trace log can be formatted into a readable trace report and sent to the display screen, a printer, or another file.

Error Loggin

Error logging is automatically enabled when you initialize the system. It can be disabled if necessary, but it usually runs as a background process, collecting error entries generated by software components. The error entries are stored in an error log that can be formatted into a readable error report. As with the trace facilities, you can use special subroutines to generate error entries from your own programs.

Dum

The system supports a dump facility that allows you to dump data onto a dump diskette in the event of a system failure. In order to start a dump operation, you need to learn how to use the dump key sequence. To examine the contents of a dump diskette, use the **crash** command. For additional information on the **crash** command, refer to the *AIX Operating System Commands Reference*.

11.4 Using the Trace Facilities

The trace facilities are used to monitor changes to variable data within software components. Tracing can be used as a debugging aid and to check the performance of various sections of code. Some of the basic trace terms are defined below:

trace entry A data structure containing a header of identifying information plus up to 20 bytes of defined data. Trace entries are generated by trace points and written to a trace log file that can be formatted by the trace formatter.

trace point A group of code statements that generates a trace entry from within a software component. Trace points are assigned to an event class which can be active or inactive. Trace points within active event classes are able to generate trace entries.

event class A number assigned to a group of trace points that relate to a specific subject or system component. The defined event classes are listed in the trace profile, **/etc/trcprofile**.

hook ID A unique number assigned to a specific trace point. All trace entries include the hook ID of the originating trace point in the trace entry header. Pre-defined trace points use assigned hook IDs ranging from 0 to 299. User-defined trace points can choose hook IDs ranging from 300 to 399.

How you use the trace facilities depends on what you want to accomplish:

To choose the event classes that you want to trace, you should lear how to alter the default trace profile, **/etc/trcprofile**, and how to create your own trace profile.

To start a trace session, you should use the **trace** command.

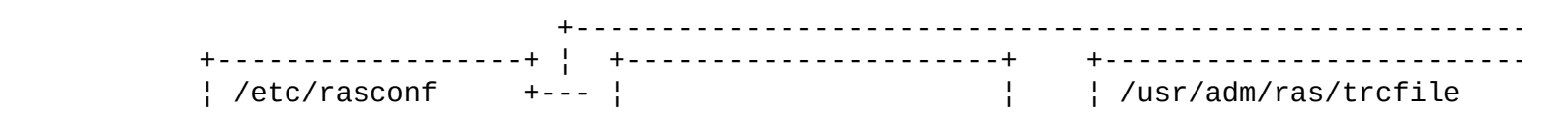
To end a trace session, you should use the **trcstop** command.

To format a **trace log** file, you should learn how to use the **trcrpt** command.

To change the name or size of the default trace log file, you shoul learn how to alter the **/etc/rasconf** file.

To create trace points that generate trace entries, you should lear how to use the trace subroutines: **trcunix**, **trsave**, and **_trcgen**.

If you create your own trace entries, you should learn how to create trace template for each type of trace entry. You also need to learn how to use the **trcupdate** command to add trace templates to the template file, **/etc/trcfmt**.



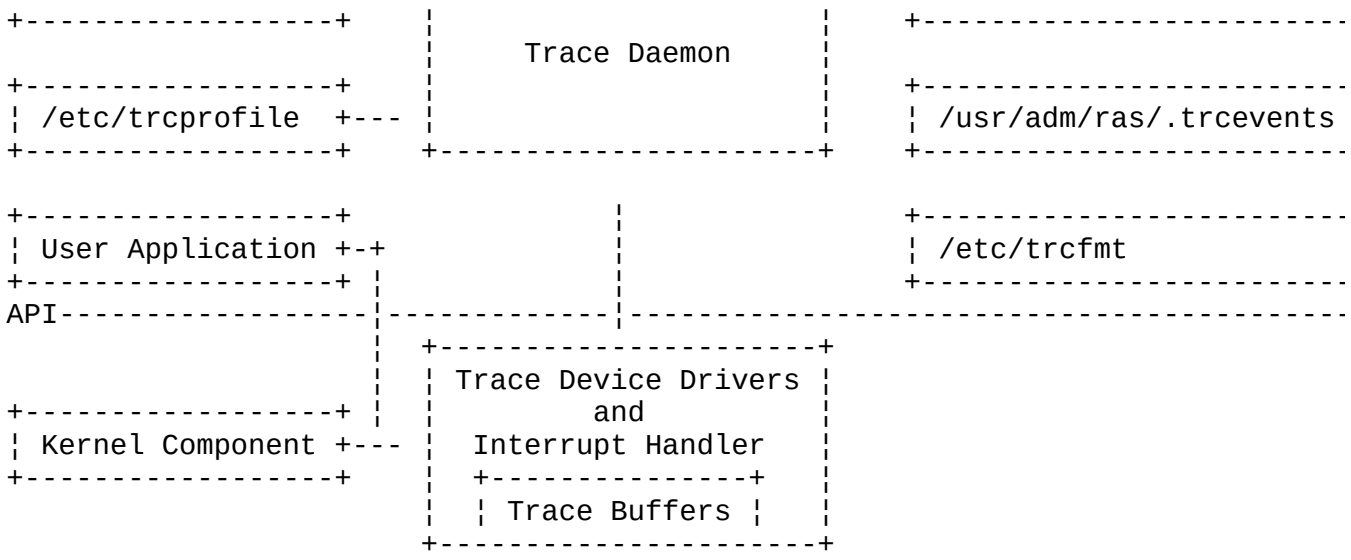


Figure 11-1. Trace Components

Figure 11-1 shows how data is passed between the various files and components that constitute the trace facilities. The lines labeled **API**, which stands for Application Program Interface, shows the logical distinction between application programs and kernel components.

The components and files in Figure 11-1 are described on the next page. They are explained in more detail elsewhere in this chapter in the appropriate sections.

The following descriptions start at the point where the trace entries are first generated and end where the trace entries are formatted and sent to standard output.

Applications and kernel components generate trace entries using trac points placed at strategic places in the execution path. The system programs contain several pre-defined trace points relating to various event classes.

The trace device drivers collect the trace entries in trace buffers There is one trace buffer for each of the three trace subroutines.

The **/etc/rasconf** file contains configuration data. For the trace daemon, it defines the name and size of the **trace log** file to be opened to receive trace entries. For the trace formatter, it defines the default **trace log** file if none is specified when it is invoked.

The default trace profile is **/etc/trcprofile**. This file contains a list of the defined event classes. An event class is either active or inactive. If an event class is active, the trace points related to that event class will generate trace entries. You can use the default trace profile, or create your own trace profile.

The trace daemon is an important part of the trace facilities. Whe it is initialized, it performs three major tasks:

- Reads the trace profile to determine which event classes should be active.

- Opens the file specified in the trace stanza in **/etc/rasconf** as

the **trace log** file.

- Begins reading the trace buffers as they become full and writes them out to the **trace log** file.

The default **trace log** file is **/usr/adm/ras/trcfile**. This file stores all of the trace entries generated by software programs. If the **trace log** file becomes full, the newest trace entry overwrites the oldest trace entry.

The **/usr/adm/ras/.trcevents** file contains lists of event classes and the hook IDs associated with those event classes. The ***hook ID*** is a specific number associated with a particular trace point. The trace formatter uses information in this file to count the trace entries that occur for each event class. You do not edit this file directly. It is automatically updated when you use the **trcupdate** command.

The trace format file, **/etc/trcfmt**, contains trace templates that determine how each trace entry appears when it is formatted. The pre-defined trace entries also have pre-defined templates. If you generate trace entries from your own programs, you need to define trace templates for those entries.

The trace formatter formats the trace entries in a trace log file int a readable format. If a **trace log** file is not specified when the trace formatter is invoked, it uses the file specified in **/etc/rasconf**. The formatted trace entries are sent to standard output and can therefore be sent to the display screen, a file, or a printer.

Subtopics

11.4.1 Altering the Trace Configuration Files

There are two basic files that you can alter to change the operation of the trace facilities. The first is the ***trace profile***, which is used by the trace daemon to set up three bit masks that show which event classes are active. The trace daemon reads the trace profile when it is initialized. A **trace profile** contains a single line entry for each defined event class. Each line entry begins with a * (asterisk). Only those event classes that have had the * removed will be able to generate trace entries during a trace session.

You edit a **trace profile** using a standard text editor. You can create several trace profiles, each designed for different tracing needs. When you run **trace**, you can specify which **trace profile** it should use. If you do not specify a **trace profile**, it uses the default **trace profile**, **/etc/trcprofile**. To create your own trace profile, copy the default **trace profile** into your current directory and edit the new copy.

If you have defined trace points in your own programs, you need to have the trace daemon activate the **User-defined Events** event class. It appears in the **trace profile** as:

```
*     150 User-defined Events
```

To have the trace daemon activate this event class, remove the * from the beginning of that line in the **trace profile**. It should now appear as:

```
      150 User-defined Events
```

If you do not enable this event class, your trace points will not be able to generate trace entries in the **trace log** file.

The other file that you can alter is the **/etc/rasconf** file. This is a configuration file that is read by the trace daemon and the trace formatter. It contains various types of information, but the information you are interested in is the stanza that defines the name and maximum block size of the default **trace log** file. It appears in **/etc/rasconf** as:

```
/dev/trace:
    file = /usr/adm/ras/trcfile
    size = 80
```

Other users may change the contents of this file. You may want to change the **trace log** file name if the current **trace log** is full, or if you want to keep different logs for different trace sessions.

Each time **trace** is invoked, the trace daemon checks **/etc/rasconf** to see which file it should open to receive trace entries. All trace entries generated during the trace session are directed to that file. When you use the trace formatter, it uses the file name in **/etc/rasconf** as the default **trace log** file if a **trace log** file is not specified. If you have several **trace log** files, you can format up to 10 at a time by invoking the trace formatter with a list of file names.

11.4.2 Using the Trace Commands

There are four commands associated with the trace facilities. They each perform one basic task:

trace Starts a trace session. This command accepts a **trace profile** as an input file name parameter. Once a trace session is initiated, all trace points with active event classes will generate trace entries if their particular component is run.

If you do not specify a **trace profile**, the default **trace profile**, **/etc/trcprofile**, is used. The following example invokes **trace** with a user-defined **trace profile** Note that an & (ampersand) is required at the end of the command line. This causes **trace** to run as a background process:

```
trace /u/myfile/myprof &
```

trcstop Stops a trace session. Any trace entries remaining in the trace buffers are written out to the **trace log** file. Any open files are closed and the trace daemon is terminated. This command does not require any input parameters.

trcrpt Formats trace entries contained in **trace log** files. This command accepts 1 to 10 **trace log** file names as input parameters. If you do not specify a **trace log** file, it uses the default **trace log** file specified in **/etc/rasconf**. The formatted trace entries are sent to standard output.

The following example formats two **trace log** files named **trace1** and **trace2** and sends the output to the printer:

```
trcrpt trace1 trace2 | print
```

Because all trace entries are time-stamped, the trace formatter can also format a subset of a **trace log** file consisting of trace entries that fall within a certain time interval. You specify a starting time and end time using the **-s** and **-e** flags. The time is specified as *MMddhhmmyy* (month, day, hour, minute, year).

The following example formats the default **trace log** entries ranging from January 3, 1985 at 11 a.m. to 11:06 a.m. of the same day:

```
trcrpt -s0103110085 -e0103110685
```

trcupdate Adds, updates, or deletes trace templates in the **/etc/trcfmt** file. Also updates the **/etc/.trcevents** file. Before using this command to add or update trace templates, you need to create an input file with the extension **.trc**. This file will contain two types of entries:

Template definitions. These contain a + (plus sign) in the first column to specify that the template is to be merged into the master template file. A single template definition may require several lines. The + is only required on the first line of each template.

Template deletions. These contain a - (minus sign) in the first column, one blank, and the hook ID of the template you want to delete.

Note: Do not delete the pre-defined trace templates (hook IDs 0 to 299). These are required to format the pre-defined trace points imbedded in the system programs.

The following example shows a file that will add trace template 330 to the master template file and delete templates 320 and 321. Note that the first line in the file must be entered as shown. It is used by the **trcupdate** command to verify that this file contains template information.

```
* /etc/trcfmt
+ 330 1.0 InitPtr Printer PtrNode A8: \n: \t: \
          PtrType D4: PtrActv B0.1, 1 Yes, 0 No:
- 320
- 321
```

To update a trace template, you need to create a new template with the same hook ID but with a version and release number (*VV.RR* field) greater than the version and release number of the trace template that you want to update. When **trcupdate** merges the **.trc** file into the master template file, it will replace the old template with the updated one.

The trace formatter writes the total number of trace entries attributed to each event class at the end of a trace report. If you want trace entries generated by new trace points to be counted under the **User-defined Events** event class, you must also create a file with an extension of **.evt**.

The **.evt** file contains entries that specify an event class number and any new hook IDs that you want to associate with that event class. It is merged into the **/etc/.trcevents** file when you use the **trcupdate** command. For user-defined trace entries, the event number is always 150. Thus, to add hook ID 330 to the **User-defined Events** event class, you would create a file as shown below. Note that the first line in the file must be entered as shown:

```
* /ras/.trcevents
150 330
```

Once you create a file containing new or updated template definitions and/or template deletions, you can use **trcupdate** to process it. In the following example, the template definitions and deletions are contained in a file called **newtemps.trc**. The event class and hook ID lists are contained in a file called **newtemps.evt**:

```
trcupdate  newtemps
```

11.4.3 Using the Trace Subroutines

Before using the trace subroutines, you need to understand how the trace daemon uses the **trace profile** and how event classes are represented as active or inactive in the system.

After the trace daemon reads the **trace profile**, it sets up two tables in memory. Each table consists of a one-word array of flag bits. Each bit is called a ***channel*** and is named after the bit position it occupies. The most significant bit is called **channel 0** and the least significant bit is called **channel 31**. Each event class is represented by one or more of these channels. When the trace daemon reads the **trace profile** and sees that an event class should be active, it sets the appropriate channels in each channel table.

When the trace daemon has read the entire trace profile, it sends one channel table to each of the two trace device drivers. When one of these drivers is called by a trace subroutine, the device driver compares the channel number, which is a required part of the trace entry, against its own channel table. If the channel is active, it puts the trace entry in a trace buffer. If not, it does nothing; however, in each case it returns a successful return code unless there is a problem in the calling procedure.

With 32 bits available for each channel table, it is possible to have 48 different event classes. However, an event class may use the same channel in more than one table. Thus, the actual number of event classes is less than 96. The **User-Defined Events** event class uses channel 31 in each of the two channel tables. This allows you to create user-defined trace entries using any of the two trace subroutines. User applications can use the **trace_on** subroutine to see if a specific channel is enabled.

Note: If you look at the **trace profile**, you will see a number placed before each event class. This number identifies a specific event class; however, it is not a channel number. Channel numbers can only be in the range from 0 to 31.

When you use a trace subroutine, one of the input parameters contains a channel number and hook ID to identify which event class that trace entry belongs to and which trace point generated the call. This is called the ***trace ID***. Bits 0 through 4 of the trace ID contain the binary representation of the channel number. Bits 5 through 15 contain the binary representation of the hook ID. Thus, for user trace entries, bits 0 through 4 should always be set to 31 (channel 31) and the hook ID should be a number ranging from 300 to 399.

The parameters used by the trace subroutines are briefly explained in the following:

trcunix Generates a trace entry from an AIX Operating System application and requires two input parameters. The first is the address of a buffer containing the 2-byte trace ID and up to 20 bytes of trace data. The second is the length of the buffer.

trsave Generates a trace entry from an AIX Operating System kernel component and requires 3 input parameters. The first is a 2-byte trace ID. The second is the length of the data buffer. The third is the address of the data buffer.

trc_start Starts the trace daemon from a process and requires the

following parameters:

outpath Specifies the file name that the trace daemon uses to store the trace data. This file name must be different from the default file name in **/etc/rasconf**.

entsize Specifies the size in bytes for the entries that the application logs. If the value is 0, the default size is used.

numents Specifies the number of entries the kernel buffer will hold. If **numents** is not used, the buffer size is taken from **/etc/rasconf**. If the value of **numents** is 0, or if an invalid (non-numeric) argument is supplied, the trace daemon calculates the buffer size using the entry size and a default of 1024 entries.

last_only Specifies whether the trace daemon should log data continuously or only save the last buffer filled before the daemon stops. A value of 0 indicates that the trace daemon should log data continuously. A value of 1 indicates that the daemon should only save the last buffer filled before the daemon stops.

structure t_struct
Contains the following control values returned by the daemon:

Process ID of the daemon

Address of the trace buffer

Length of the trace buffer

Channel mask used by the process when logging trace data. The channel mask is the decimal number of the bit position assigned to the trace channel, shifted left 11 bits.

trc_stop Stops the trace daemon from the process ID started by **trc_start**.

When you create your own trace points, remember that each trace point should use a unique hook ID in the range from 300 to 399. You must also create trace templates for your trace points. Here are some general guidelines:

Put a trace point at the beginning or end of an important function The trace data should show the values of any important data structures or I/O parameters.

If a function takes a significant amount of processing time, you ma want to put a trace point at its entry and exit points. This will

allow you to trace how long it takes to process different types of input parameters.

Put a trace point at a critical junction in the logic flow of component.

Do not put a trace point inside a loop that is repeated many times i you can catch the important data before the loop begins.

Try to limit the trace entry data to four variables or less

The trace device drivers have access to the channel tables, so they know whether or not to put a trace entry in the trace buffers. However, you may want to explicitly check to see if your channel is active before you call a trace subroutine so you can design the trace point to skip around data collection code if the channel is not on. For kernel components, the usual procedure is to perform the subroutine call without checking for an active channel because kernel trace points do not contain much data.

A sample trace point for application level components is shown in Figure 11-2. Notice how it uses the **trace_on** subroutine to see if the **User-Defined Events** channel (31) is on before calling the **trcunix** subroutine. Also, notice how the trace ID is created. First, you shift the variable containing the channel number left 11 bits so that it is in bits 0 through 4. Then you perform an OR operation on that variable with the variable containing the hook ID. The result is a trace ID containing the channel number in bits 1 through 4 and the hook ID in bits 5 through 15.

```
#include <sys/trace.h>
#include <trcdefs.h>
#include <stdio.h>

main(argc, argv)
   int argc;
   char **argv;
{
   extern int trace_on();
   extern int trcunix();
   int chan_no = 31;
   long channel = 0x1;   /* Set bit 31 (user-defined channel) to 1 */
   long hookid = 300;    /* Define the hook ID for this trace point */
   int tracing;
   int rc = 0;
   struct
   { unsigned short traceid;
      char data[20];
   } trcstruct;

   /* See if channel 31 is active.  */

   if ( (tracing = trace_on(channel) ) < 0)
   {
      fprintf(stderr,"trace_on failed\n");
      return(-1);
   }

   /* Imbed this trace point anywhere in your program code. */
```

```
    if (tracing)
    {
       trcstruct.traceid = ((chan_no<<11) | hookid);
       sprintf(trcstruct.data, "This is trace data.");
       if (trcunix(&trcstruct, sizeof(trcstruct)) < 0)
       {
          fprintf(stderr,"trcunix failed\n");
          rc = -1;
       }
    }
    exit(rc);
 }   /* end main */
```

Figure 11-2. Example Program Fragment Showing Use of trcunix Subroutine

Figure 11-3 creates a trace entry from within an AIX Operating System kernel component. Notice how it uses a defined constant (TR_USER) to set the channel number in the trace ID parameter. TR_USER contains channel number 31 in bits 0 through 4. This is only available for kernel components using the **trsave** routine.

```
 #include <sys/trace.h>
 #define TRACEDATA 1

 function()
 {
    long hookid = 310;
    int t_data;

    /* The following trace point can appear anywhere in your function. */

    t_data = TRACEDATA;
    trsave((TR_USER | hookid), sizeof(int), &t_data);
 }   /* end function */
```

Figure 11-3. Example Program Fragment Showing Use of trsave Subroutine

11.4.4 Creating Trace Templates

The trace formatter uses ***trace templates*** to determine how the data contained in trace entries should be formatted. All trace templates are stored in the master template file, **/etc/trcfmt**. After you create some new or updated trace templates, place them in a file that can then be merged into the master template file using the **trcupdate** command. The **trcupdate** command requires that trace templates identify themselves by placing a + and a blank in the first two columns of the first line of the trace template.

Trace templates contain four required fields and zero or more data description fields. The required fields identify the template hook ID, the version and release number, the hook ID label, and the event class label. The data description fields contain formatting information for the trace entry data and can be repeated as many times as is necessary to format all of the trace data in the trace entry.

Trace entries are formatted and written to standard output one entry at a time. For each entry, the trace formatter performs the following operations:

1. Locates the trace template corresponding to the hook ID in the trace entry.

2. Writes the time the entry was generated. It also writes a sequence number that shows when the entry was generated as an interval within a second. This sequence number allows the formatter to sort trace entries that are generated during the same second. They can then be formatted and written in the order in which they occurred. These fields are placed in the trace entry header by the trace subroutine.

3. Writes the PID contained in the trace entry header. These fields are placed in the trace entry header by the trace subroutine.

4. Writes the hook ID and event class. You can use the actual numbers that equate to the hook ID and event class, or you can create a name up to eight characters for each field. The trace formatter will write whatever is placed in the field.

5. Writes up to 20 bytes of data according to the *data_descriptors* in the trace template, if any are present. Any data that occurs after the last *data_descriptor* in the template is ignored. A *data_descriptor* does the following:

    a. Writes the data label (*d_label*), if it is specified.

    b. Writes the actual data according to the *format* field and the optional match fields. You can define match fields that will replace data values with descriptive labels or change to a different *data_descriptor* depending on the current data value.

Figure 11-4 shows the syntax used to define a trace template. Figure 11-5 defines each of the fields in a trace template.

Trace template lines can be as long as you need. You can continue a trace template on another line by adding the \ (backslash) character to the end of the line where the split occurs. Note that \ is not a substitute for a required blank. A blank is also required after a colon or comma.

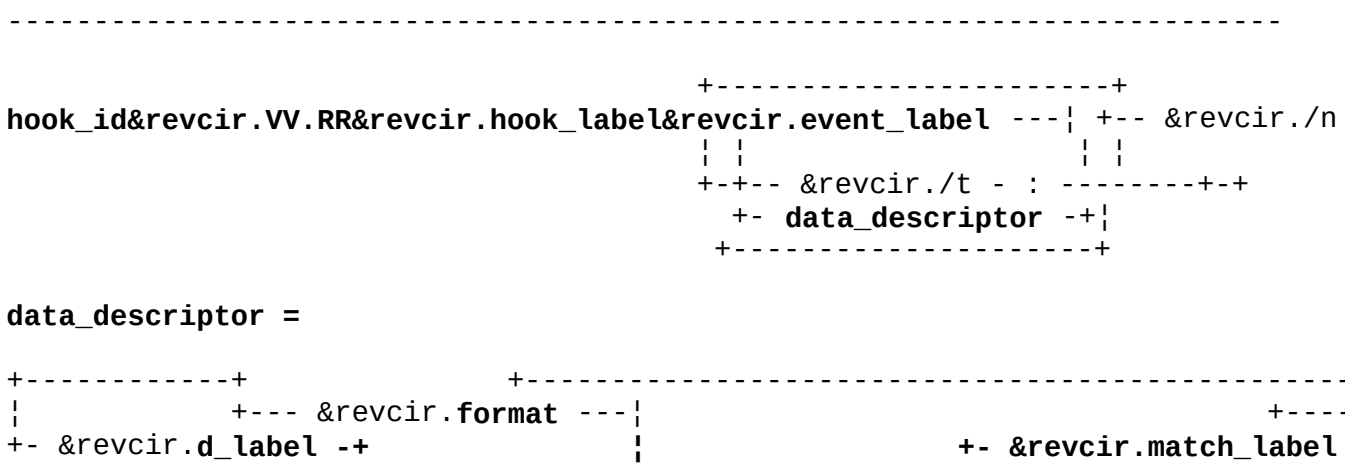


Figure 11-4. Trace Template Syntax

Figure 11-5. Fields in a Trace Template

Field	Description
hook_id	The 3-digit hook ID of the trace entries that use this template.
VV.RR	A number representing the version and release level of this template.
hook_label	A label field of up to 8 characters that describes the hook ID.
event_label	A label field of up to 8 characters that describes the event class.
\n	A special string used to start output data on a new line.
\t	A special string used to start output data after one tab unit.
d_label	A label field of up to 8 characters that describes the trace entry data.
format	An alphanumeric code that defines the format of the trace data.

Code	Format of Data
A*m*	ASCII string of *m* characters.
B*m*	Binary string of *m* bytes.
B*m.n*	Binary string of *m* bytes and *n* bits. If this format leaves some unformatted bits in the current byte, they are not written unless this format is followed by another format that specifies bits.
D2	Decimal short integer.
D4	Decimal long integer.
F4	Floating-point number of type *float*, rounded to four places.
F8	Floating-point number of type *double*, rounded to four places.

O*m*	Omit (do not write) the next *m* bytes.
O*m.n*	Omit (do not write) the next *m* bytes and *n* bits. If this format leaves some unformatted bits in the current byte, they are not written unless this format is followed by another format that specifies bits.
U2	Unsigned decimal short integer.
U4	Unsigned decimal long integer.
X*m*	Hexadecimal number of *m* bytes.

match_value A value with a data type the same as the *format* field. If you specify a *match_value*, you must also specify a *match_label* or *data_descriptor*.

match_label A character field up to 120 characters that replaces output data that matches a *match_value* field.

data_descriptor A field containing formatting information for a portion of the output data.

Subtopics

11.4.4.1 Replacing Values in the Output Data

The *match_value* field provides a function that is similar to a switch statement in programming. You can use this function to do either or both of the following:

Replace an output data value with a character string that describe the data. For example, you can replace a numeric error code with a description of that error. The replacement character string is called a *match_label*.

Change the data descriptor used to format the output data. Fo example, you can design trace points that will record different data at different times, depending on the state of your machine or program. You can then define a set of *match_values* with a different *data_descriptor* for each type of data.

You can specify any number of *match_value* fields for a particular output data value as long as each field is accompanied by a *match_label* and/or a *data_descriptor* field. If the output data value matches *match_value*, the trace formatter uses the *match_label* and/or *data_descriptor* associated with the *match_value*.

You can use the special string ***** as a *match_value*. This will match any output data value in the current data field. When the trace formatter finds this string, it does not check any *match_values* that occur after this string. Therefore, you should use this string as the last *match_value* in a list to provide a default action if none of the other *match_values* are matched. If no match is found, the data value is written.

If a new *data_descriptor* follows the *match_value*, the trace formatter uses the formats from the new *data_descriptor* until one of the following occurs:

It has formatted all of the trace entry data

It has used all of the defined *data_descriptors* for the current output data.

The trace formatter then looks for a new *data_descriptor* field to define the format for the rest of the data in the trace entry. If a new *data_descriptor* is not found, it writes the data literally according to the format given in the *format* field of the last *data_descriptor*.

Note: The trace formatter compares a *match_value* field and the current output data as two character strings. For real numbers (*F* format), it uses the **printf** subroutine to round the data to four decimal places. If you plan to use a *match_value* field with output data formatted as a real number, the *match_value* field should use the same precision, four decimal places.

11.4.4.2 Appearance of the Formatted Output Data

The following examples show you how to format data using a single *data_descriptor*. The data is defined as 8 bytes consisting of **IBM4341A**. The *data_descriptor* fields have the following values:

d_label	**PtrNode**
format	**A8**
match_value	**IBM4341A**
match_label	**Model4X**

Each of the following examples shows a *data_descriptor* using a combination of *data_descriptor* fields and how the output data would look if it used that *data_descriptor*:

If you just specify a *format* field, the *data_descriptor* would be:

```
A8:
```

The formatted trace report would show the data as:

```
IBM4341A
```

If you specify *d_label* and *format* fields, but do not specify a *match_value* field, the *data_descriptor* would be:

```
PtrNode A8:
```

The formatted trace report would show the data as:

```
PtrNode=IBM4341A
```

If you specify *format*, *match_value*, and *match_label* fields, but do not specify a *d_label* field, the *data_descriptor* would be:

```
A8, IBM4341A Model4X:
```

The formatted trace report would show the data as:

```
Model4X
```

If you specify *d_label*, *format*, *match_value*, and *match_label* fields, the *data_descriptor* would be:

```
PtrNode A8, IBM4341A Model4X:
```

The formatted trace report would show the data as:

```
Ptrnode=Model4X
```

11.4.4.3 Trace Template Example

This section shows you how a trace entry using a sample trace template would appear after being formatted. In the example, a trace point belonging to the printer event class generated a trace entry during printer initialization. The header data for the trace entry includes the hook ID, which in this example is 330. The trace entry data is listed below:

PtrNode The printer node ID (8 ASCII characters). The *format* for this data would be A8.

PtrType The printer type (a decimal long integer). The *format* for this data would be D4.

PtrActv A 1-bit flag that indicates if the printer is active (1) or inactive (0). The *format* for this data would be B0.1.

Figure 11-6 shows a sample trace template for trace entries with a hook ID of 330. Notice the newline and tab descriptors before the **PtrType** field. This will cause the **PtrType** field to appear on the next line and indented one standard tab unit. Also, notice that two pairs of *match_value* and *match_label* fields are defined for the 1-bit **PtrActv** flag. This tells the trace formatter to write **Yes** if the flag equals **1** and **No** if the flag equals **0**:

```
-------------------------------------------------------------------------

  330 1.0 InitPtr Printer PtrNode A8: \n: \t: \
          PtrType D4: PtrActv B0.1, 1 Yes, 0 No:


-------------------------------------------------------------------------
```

Figure 11-6. Example of a Trace Template for hook ID 330

Figure 11-7 in topic 11.5 shows how a trace entry with a hook ID of 330 might appear in a formatted trace report. In the report, the sample trace entry is the seventh formatted entry. Notice that the last part of the report shows a list of event classes and how many trace entries belong to each event class.

11.5 Using the Error Log Facilities

The error log facilities are used to record errors that may occur in the system. These errors can be in hardware or software, and can be of several different types. Some of the basic error terms are defined below:

error entry A data structure containing a header of identifying information plus several bytes of defined data. Error entries are generated by error points and written to an error log file that can be formatted by the error formatter.

error point A group of code statements that generates an error entry from within a software program. Error entries are generated when a software or hardware component encounters an error.

error ID This is part of the data required by an error entry. It is a unique combination of three hexadecimal digits that identifies the component that generated the error entry.

error identifier
A three-character code used to identify error templates and to specify which error entries the error formatter should process. This code is based on the error ID; however, it use alphanumeric characters instead of hexadecimal digits.

```
                                TRACE LOG REPORT

  File: /usr/adm/ras/trcfile

  Fri Jan  3 13:19:35 1985
  System: ****            Node: ****

  Version: 0                    Machine: ****

  TIME         SEQ   PID  IODN  IOCN TYPE     HOOK     DATA

  23:23:34.01 0001 00195            I/O_Sys  ioctl[x] fildes=5 request=23
         arg=536872136
  23:23:34.01 0002 00195            I/O_Sys  ioctl[x] fildes=1 nbyte=28
  23:23:34.01 0003 00195            I/O_Sys  write[x] fildes=1 nbyte=40
  23:23:34.01 0004 00195            I/O_Sys  write[x] fildes=1 nbyte=31
  23:23:34.01 0005 00142            I/O_Sys  write[x] fildes=2 nbyte=1
  23:23:34.01 0006 00142            I/O_Sys  write[x] fildes=2 nbyte=2
  23:23:34.01 0007 00140            Printer  Initptr  PtrNode=IBM4341
         Ptrtype=3 PtrActv= No
  23:23:34.01 0008 00196            I/O_Sys  read[x]  fildes=3 nbyte=128
  23:23:34.01 0009 00196            I/O_Sys  access   errno=2 filemode=1
  23:23:34.01 0010 00196            I/O_Sys  access   errno=2 filemode=1
  23:23:34.01 0011 00196            I/O_Sys  ioctl[x] fildes=0 request=17
         arg=1073737112

  Summary of event counts.
  Event 0: 2            Event 66: 8
  Event 150: 1
```

```
  Total number of events: 3
```

Figure 11-7. Example of Output from the Trace Formatter

How you use the error facilities depends on what you want to accomplish:

To start the error log facilities, you should use the **errdemon** command. You usually will not have to use this command because the default initialization process is set to turn error logging on.

To stop error logging, you should use the **errstop** command. You must have superuser authority to use this command.

To format the error log files, you should learn how to use the **errpt** command.

To change the name or size of the default error log files, you shoul learn how to alter the **/etc/rasconf** file.

To create error points that generate error entries, you should lear how to use the error subroutines: **errunix** and **errsave**.

If you create your own error entries, you should learn how to creat an error template for each type of error entry. You also need to learn how to use the **errupdate** command to add error templates to the template file, **/etc/errfmt**.

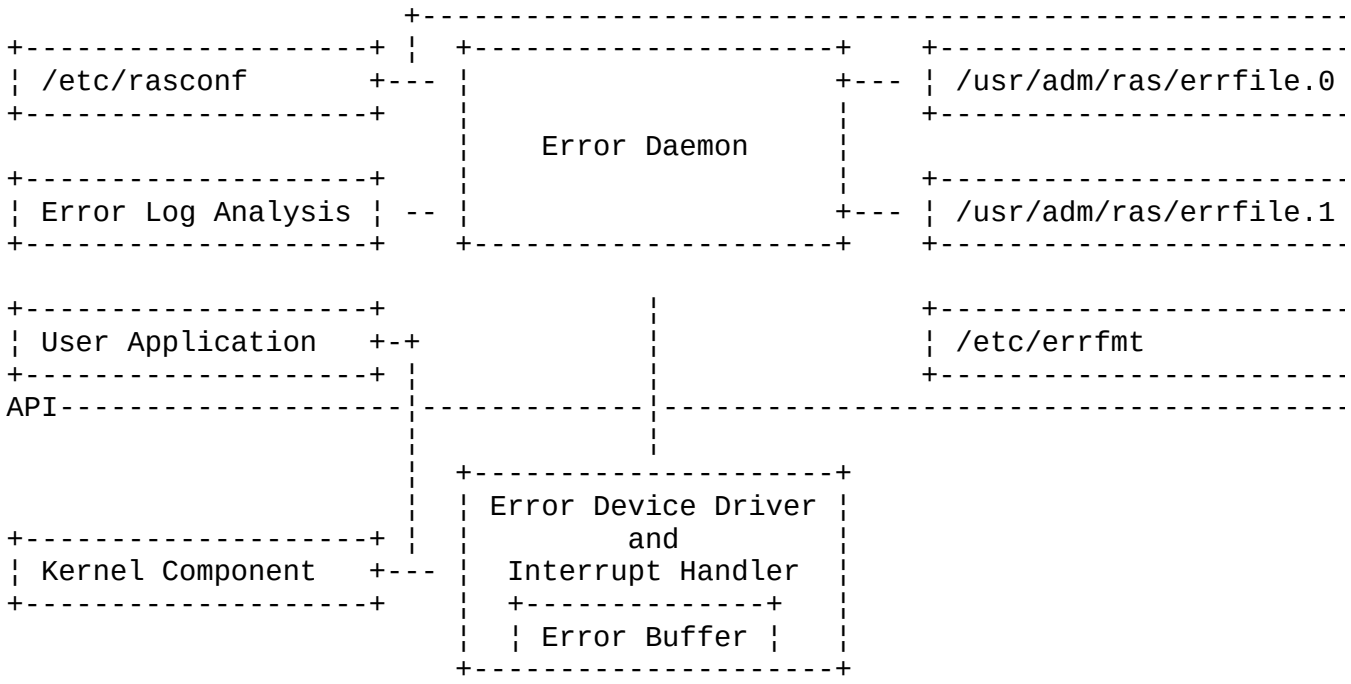


Figure 11-8. Error Components

Figure 11-8 shows how data is passed between the various files and components that constitute the error facilities. The lines labeled **API**,

which stands for Application Program Interface, show the logical distinction between application programs and kernel components. The lines connecting the files and components show where data comes from and where it goes.

The components and files in Figure 11-8 are described on the next page. They are explained in more detail elsewhere in this chapter in the appropriate sections.

The descriptions below start at the point where the error entries are first generated and end where the error entries are formatted and sent to standard output.

Applications and kernel components generate error entries using erro points. The system programs contain several pre-defined error points that generate error entries when necessary.

The error device driver collects the error entries in the erro buffer. There is one error buffer that collects error entries from all three error subroutines.

The **/etc/rasconf** file contains configuration data. For the error daemon, it defines the name and size of the file that the error daemon uses to open two error log files. For the error formatter, it defines the default error log file name if a file name is not specified when it is invoked.

The error daemon is an important part of the error facilities. Whe it is initialized, it performs the following major tasks:

- It opens two error log files by appending the extensions **.0** and **.1** to the file name specified in the error stanza in **/etc/rasconf**.

- It checks the non-volatile RAM and if there is data, generates an error entry.

The default error log files are **/usr/adm/ras/errfile.0** and **/usr/adm/ras/errfile.1**. These files are organized as a circular buffer and store all of the error entries generated by the software programs. If one of the error log files becomes full, the next error entry is written into the other file, discarding any entries that may have been in the file.

The error format file, **/etc/errfmt**, contains error templates that determine how each error entry appears when it is formatted. The pre-defined error entries also have pre-defined templates. If you generate error entries from your own programs, you need to define error templates for those entries.

The error formatter formats the error entries in the error log file into a readable format. If an error log file is not specified when the error formatter is invoked, it uses the file specified in **/etc/rasconf**. It reads the error files and sends error entries to the error log problem determination program, **errpd**. The **errpd** program analyzes the error entries and returns probable cause information to the error formatter. When **errpd** is finished, the error formatter appends the **errpd** analysis information, if any, onto the error entry and formats the entry. The formatted error entries are sent to standard output and can therefore be sent to the display screen, a

file, or a printer.

Subtopics

11.5.1 Altering the Error Log Configuration File

There is one file that you can modify before starting the error daemon. This is the **/etc/rasconf/** file, which is a configuration file that is read by the error daemon and the error formatter. It contains various types of information, but the information you are interested in is the stanza that defines the name and maximum block size of the default error log file. It appears in **/etc/rasconf** in the following form:

```
/dev/error:
    file = /usr/adm/ras/errfile
    size = 50
```

Some other use may change the contents of this file. You may want to change the error log file name if the current error log files are full, or if you want to keep different logs for different types of error entries.

Each time **errdemon** is invoked, the error daemon checks **/etc/rasconf** to see which files it should open to receive error entries. The error daemon does not open the actual file name specified in **/etc/rasconf**. It merely uses the file name as the basis for two other file names with extensions of **.0** and **.1**. The file names with the extensions are the ones that are actually opened. For example, if the error stanza in **/etc/rasconf** is:

```
/dev/error:
    file = /usr/adm/ras/errfile
    size = 50
```

the error daemon will open two files using the names:

```
/usr/adm/ras/errfile.0
/usr/adm/ras/errfile.1
```

All error entries generated while the error daemon is active are directed to these two files. When you use the error formatter, it uses the file name in **/etc/rasconf** as the default error log file if an error log file is not specified. The error formatter adds the **.0** and **.1** extensions to the error log file name to determine the actual names of the two error log files.

11.5.2 Using the Error Log Commands

There are four commands associated with the error log facilities. They each perform one basic task:

errdemon Starts an error session. Once the error daemon is initiated, all error points will generate error entries whenever their error conditions are met. This command is usually placed in the command file **/etc/rc**, which is run when the system is initialized. This command does not require any input parameters.

errstop Stops the error daemon. Any error entries remaining in the error buffer are written out to the error log file. Any open files are closed and the error daemon is terminated. This command does not require any input parameters. Note that it is not necessary to stop the error daemon in order to format the error log files.

errpt Formats error entries contained in error log files. This command accepts one or more error log file names as input parameters. If you do not specify an error log file, it uses the default error log file specified in **/etc/rasconf**. The formatted error entries are sent to standard output.

The following example formats two error log files named **myerr.0** and **myerr.1** and sends the output to the printer. Notice that you use the file name **myerr** as input. The error formatter automatically appends the extensions and looks for those files, not the file named in the command:

```
errpt myerr | print
```

There are several options available to format subsets of the entire error log. See the description of the **errpt** command in *AIX Operating System Commands Reference*.

errupdate Adds, updates, or deletes error templates in the **/etc/errfmt** file. Before using this command to add or update error templates, you need to create an input file with the extension **.err**. This file will contain two types of entries:

Template definitions. These contain a + in the first column to specify that the template is to be merged into the master template file. A single template definition may require several lines. The + is only required on the first line of each template.

Template deletions. These contain a - (minus sign) in the first column, one blank, and the error identifier of the template you want to delete.

Note: Do not delete the pre-defined error templates. The only templates that you should consider deleting are those that are user-defined. User-defined error templates have an error identifier beginning with **U** or **HF**.

The following example shows a file that will add error template U13 to the master template file and delete templates U11 and U12. Note that the first line in the file must be entered as shown. It is used by the **errupdate** command to verify that this file contains template information.

```
* /etc/errfmt
+ U13 Serial/Parallel Adapter: \n: \
      ErrorType A1: -n: LastI/O X1: \
      LineStatus D4: PrinterStatus B0.1:
- U11
- U12
```

To update an error template, you need to create a new template with the same error identifier but with a version and release number (*VV.RR* field) greater than the version and release number of the error template that you want to update. When **errupdate** merges the **.err** file into the master template file, it will replace the old template with the updated one.

Once you have created a file containing new or updated template definitions and/or template deletions, you can use **errupdate** to process it. In the following example, the template definitions and deletions are contained in a file called **newtemps.err**:

```
errupdate  newtemps
```

11.5.3 Using the Error Log Subroutines

To use the error log subroutines, you need to understand the relationship between error IDs and error identifiers. You also need to know how to identify the type of an error.

Error IDs are 3 bytes long, and are created using hexadecimal digits. When you use the error log subroutines, you must specify the error ID of the program that generates the error entry. The pre-defined error IDs can range from 0x010000 to 0x050F0F. User-defined error IDs can range from 0x060000 to 0x060F0F. There is a special set of user-defined error IDs for hardware error entries. These error IDs can range from 0x010F00 to 0x010F0F. Note that the second and third bytes cannot use values greater than 0x0F for any error ID.

The error ID is used to categorize the error entries into appropriate groupings. The first digit is the ***class***; the second is the ***subclass***, and the third is the ***mask***. The digits are hierarchical in that a class consists of zero or more subclasses and a subclass consists of zero or more masks.

The class digits range from 0x01 to 0x06. Each value represents a general category of errors. The categories are assigned as follows:

```
0x01 -- Hardwar
0x02 -- Softwar
0x03 -- IPL/Shutdow
0x04 -- General System Conditio
0x05 -- Not Availabl
0x06 -- User-Defined
```

The subclass and mask digits range from 0x00 to 0x0F. They are used to further subdivide the category represented by the class digit. Classes 0x01 through 0x05 may have pre-defined subclasses and masks. Class 0x06 has no pre-defined subclasses or masks because it is user-defined.

When you format a file containing error entries, you have the option of specifying which class of errors should be formatted. You can also specify the subclass within the class and the mask within the subclass. Note that when you format error entries, you do not use the error ID. Instead, you use an ***error identifier***. Error identifiers are similar to error IDs, except they consist of characters instead of hexadecimal digits and an alphabetic letter is used to specify the class instead of a digit. The letter is the first letter of the first word describing the class.

For example, if the error ID of an error entry is 0x010102, the error entry is in the Hardware class. The error identifier for this type of error entry would be H12. An error entry in the Software class might have an error ID of 0x020303. The error identifier would be S33. Thus, An error entry in the User-defined class would have an error identifier that began with the letter U.

Note: You should avoid defining error IDs that have a subclass or mask value of 0x00. This is because the error formatter uses the character 0 as a pattern-matching character. For example, to format all hardware error entries, regardless of subclass or mask, you would specify **H00** when you invoke the **errpt** command. The use of 0 within an error ID limits your selection options when you format the error entries.

When you design a program, you must decide what constitutes an error

condition and what actions should take place if an error does occur. You decide in advance what the error ID will be for any error entries generated by a specific component. You must also specify the type of error entry that is being generated. There are several pre-defined types of error entries. Your component must contain an algorithm for deciding which type of error has occurred. Once the type is identified, it is used as input data to the error log subroutines.

The types of error entries are defined as follows:

1 -- Permanent
These are errors that are severe enough to prevent successful completion of an operation.

2 -- Temporary
These are errors that require an operation to be retried a defined number of times before being successfully completed.

3 -- Information
These are not necessarily errors. A component may generate an error entry of this type if an unusual condition occurs.

4 -- Counters
These error entries are generated by device driver components. Certain device drivers are able to generate retries if an operation is not successful on the first attempt. They use counters to monitor the number and cause of retries and contain algorithms that decide when these counters should be sent to the error log.

5 -- Abbreviated Error Entries
If the system fails before storing an error entry in the error log, it writes an abbreviated entry into non-volatile storage. When the system is started following such a failure, it writes this abbreviated entry into the error log.

6 -- Expert Analysis Appended
These error entries contain information from the Error Log Analysis routine.

The data in an error entry is entirely dependent on the component generating the error entry. There is no defined limit on the size of an error entry. How you organize the information required to generate an error entry depends on which error log subroutine you use. Both **errunix** and **errsave** use an integer length field. This field specifies the length plus one (in words) of the error entry data. They are briefly defined below:

errunix Generates an error entry from an AIX Operating System application and requires two input parameters. The first is the address of a buffer containing the 3-byte error ID, the 1-byte type number, an integer length field, and the error data for the error entry. The second is the length of the buffer.

errsave Generates an error entry from an AIX Operating System kernel component and requires two input parameters. The first is the address of a buffer containing the 3-byte error ID, the 1-byte type number, an integer length field, and the error data for the error entry. The second is the length of the buffer.

A sample error point for application level components is shown in Figure 11-9. Notice how the input data structure used by the error log subroutine is organized and how the required data is placed in it. Also, the error ID for the trace entry is 0x060104. Thus, the error template used to format the error entry would use an error identifier of U14.

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

main(argc, argv)
   int argc;
   char **argv;
{
   extern int errunix();
   int error=0;
   struct error_struct   /* this is an input parameter for errunix */
   { union
      { struct
         {
            char er_class;
            char er_subclass;
            char er_mask;
            char er_type;
         } er_csmt;
         int csmt;
      } e_id;
      int e_len;       /* length + 1 of error entry data in words */
      char e_data[40];
   } error_data;

   #define CLASS     error_data.e_id.er_csmt.class
   #define SUBCLASS  error_data.e_id.er_csmt.subclass
   #define MASK      error_data.e_id.er_csmt.mask
   #define TYPE      error_data.e_id.er_csmt.type

   /* Check for errors and increment error flag if an error is found. */
   if (error_check() < 0)           /*  User-defined function */ /
     error++;

   /* if the error flag is positive, generate an error entry. */
   if (error)
   {
      CLASS    = E_USER;   /* E_USER is predefined as 0x06. */
      SUBCLASS = 0x01;
      MASK     = 0x04;
      TYPE     = E_TMP;    /* E_TMP is pre-defined as 0x40 */

      /* Collect the error data and put it in error_data.e_data */
      fill_in(error_data.e_data);   /*  User-defined function */ /

      /* Perform the subroutine call */
      if (errunix(&error_data, sizeof(struct error_struct) ) < 0)
      {
         fprintf("%s: Cannot log error. errunix failed.\n", argv[0]);
         exit(-1);
      }
   }
   .
```

```
    .
    .
  }   /* end main */
```

Figure 11-9. Example Program Fragment Using errunix Subroutine

Figure 11-10 creates an error entry from within an AIX Operating System kernel component. Note that the user-defined subroutine **fill_in** must define the error ID and error data. The method is similar to that used in Figure 11-9; therefore, it is not repeated for this example:

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

  main(argc,argv)
      int argc;
      char **argv;
  {
     struct errinfo *errptr;

     if (error)

     /* collect error data and call errsave */
     {
        fill_in(errptr);   /* User-defined subroutine)
        errsave(errptr, sizeof(struct errinfo));
     }
  }   /* end main */
```

Figure 11-10. Example of a Program Fragment Showing Use of errsave Subroutine

11.5.4 Creating Error Templates

The error formatter uses ***error templates*** to determine how the data contained in error entries should be formatted. All error templates are stored in the master template file, **/etc/errfmt**. After you create some new or updated error templates, place them in a file that can then be merged into the master template file using the **errupdate** command.

Error templates contain three required fields and zero or more data description fields. The required fields consist of the error identifier, the version and release number, and the error label. The data description fields contain formatting information for the error entry data and can be repeated as many times as is necessary to format all of the error data in the error entry.

Error entries are formatted and written to standard output one entry at a time. For each entry, the error formatter performs the following operations:

1. Locates the error template whose error identifier corresponds to the error ID in the error entry. If the error entry has an error ID of 0x010201, the corresponding error template would have an error identifier of H21.

2. Writes the time the entry was generated. This field is placed in the error entry header by the error subroutine.

3. Writes the PID contained in the error entry header. These fields are placed in the error entry header by the error subroutine.

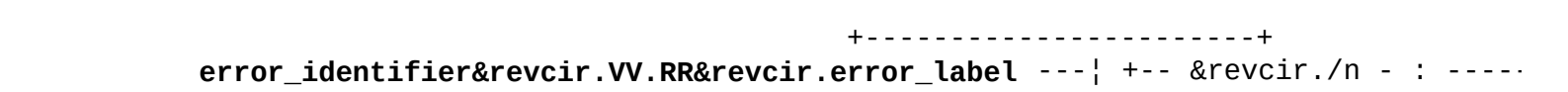
4. Writes the error label contained in the format template. You can use the actual error identifier, or you can create a name up to 8 characters. The error formatter will write whatever is placed in the field.

5. Writes the error data according to the *data_descriptors* in the error template, if any are present. Any data that occurs after the last *data_descriptor* in the template is ignored. A *data_descriptor* does the following:

   a. Writes the data label (*d_label*), if it is specified.
   b. Writes the actual data according to the *format* field and the optional match fields. You can define match fields that will replace data values with descriptive labels or change to a different *data_descriptor* depending on the current data value.

Figure 11-11 shows the syntax used to define an error template. Figure 11-12 defines each of the fields in an error template.

Error template lines can be as long as you need. You can continue an error template on another line by adding the \ (backslash) character to the end of the line where the split occurs. Note that \ is not a substitute for a required blank. A blank is also required after a colon or comma.



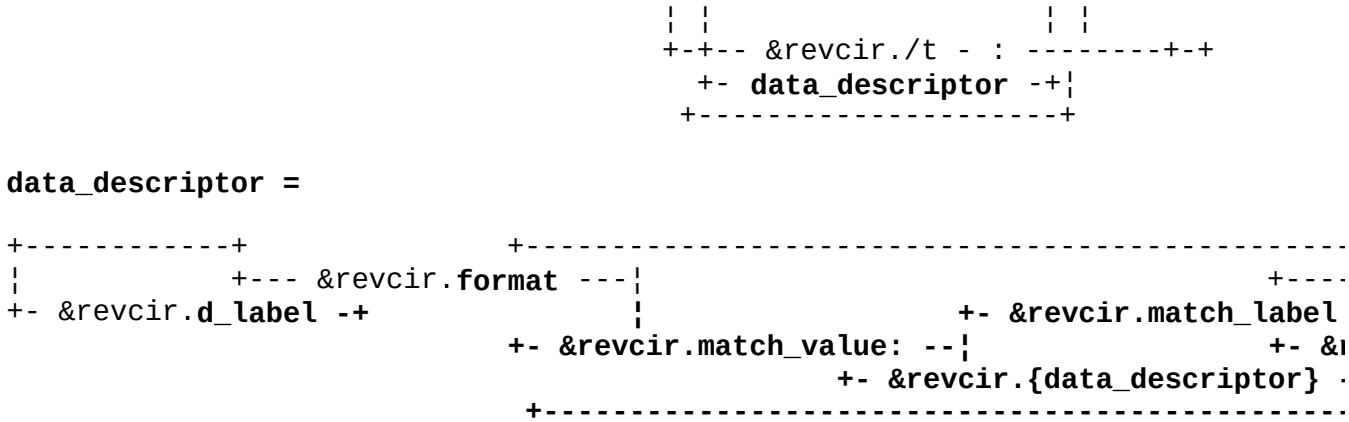


Figure 11-11. Error Template Syntax

Figure 11-12. Fields in an Error Template

Field	Description
error_identifier	The 3-character error identifier of this template.
VV.RR	A number representing the version and release level of this template.
error_label	A field of up to 14 characters that describes the error entries that use this template. This appears in the summary header list under the **Subclass** heading.
\n	A special string used to start output data a new line.
\t	A special string used to start output data after one tab unit.
d_label	A field of characters that describes the error entry data.
format	An alphanumeric code that defines the format of the error data.

Code	Format of Data
A*m*	ASCII string of *m* characters.
B*m*	Binary string of *m* bytes.
B*m.n*	Binary string of *m* bytes and *n* bits. If this format leaves some unformatted bits in the current byte, they are not written unless this format is followed by another format that specifies bits.
D2	Decimal short integer.
D4	Decimal long integer.
F4	Floating-point number of type *float*, rounded to four places.
F8	Floating-point number of type *double*, rounded to four places.
O*m*	Omit (do not write) the next *m* bytes.
O*m.n*	Omit (do not write) the next *m* bytes and *n* bits. If this format leaves some unformatted bits in the current byte, they are not written unless this format is followed by another format that specifies bits.

U2	Unsigned decimal short integer.
U4	Unsigned decimal long integer.
X*m*	Hexadecimal number of *m* bytes.

match_value	A value with a data type the same as the *format* field. If you specify a *match_value*, you must also specify a *match_label* or *data_descriptor*.
match_label	A character field up to 120 characters that replaces output data that matches a *match_value* field.
data_descriptor	A field containing formatting information for a portion of the output data.

Subtopics
11.5.4.1 Replacing Values in the Output Data
11.5.4.2 Appearance of the Formatted Output Data
11.5.4.3 Error Template Example

11.5.4.1 Replacing Values in the Output Data

The *match_value* field provides a function that is similar to a switch statement in programming. You can use this function to do either or both of the following:

Replace an output data value with a character string that describe the data. For example, you can replace a numeric error code with a description of that error. The replacement character string is called a *match_label*.

Change the data descriptor used to format the output data. Fo example, you can design error points that will record different data at different times, depending on the state of your machine or program. You can then define a set of *match_values* with a different *data_descriptor* for each type of data.

You can specify any number of *match_value* fields for a particular output data value as long as each field is accompanied by a *match_label* and/or a *data_descriptor* field. If the output data value matches *match_value*, the error formatter uses the *match_label* and/or *data_descriptor* associated with the *match_value*.

You can use the special string ***** as a *match_value*. This will match any output data value in the current data field. When the error formatter finds this string, it does not check any *match_values* that occur after this string. Therefore, you should use this string as the last *match_value* in a list to provide a default action if none of the other *match_values* are matched.

If a new *data_descriptor* follows the *match_value*, the error formatter uses the formats from the new *data_descriptor* until one of the following occurs:

It has formatted all of the error entry data

It has used all of the defined *data_descriptors* for the current output data.

The error formatter then looks for a new *data_descriptor* field to define the format for the rest of the data in the error entry. If a new *data_descriptor* is not found, it writes the data literally according to the format given in the *format* field of the last *data_descriptor*.

Note: The error formatter compares a *match_value* field and the current output data as two character strings. For real numbers (*F* format), it uses the **printf** subroutine to round the data to four decimal places. If you plan to use a *match_value* field with output data formatted as a real number, the *match_value* field should use the same precision, four decimal places.

11.5.4.2 Appearance of the Formatted Output Data

The following examples show you how to format data using a single *data_descriptor*. The data is defined as 8 bytes consisting of **IBM4341A**. The *data_descriptor* fields have the following values:

d_label	**PtrNode**
format	**A8**
match_value	**IBM4341A**
match_label	**Model4X**

Each of the following examples shows a *data_descriptor* using a combination of *data_descriptor* fields and how the output data would look if it used that *data_descriptor*:

If you just specify a *format* field, the *data_descriptor* would be:

```
A8:
```

The formatted error report would show the data as:

```
IBM4341A
```

If you specify *d_label* and *format* fields, but do not specify a *match_value* field, the *data_descriptor* would be:

```
PtrNode A8:
```

The formatted error report would show the data as:

```
PtrNode=IBM4341A
```

If you specify *format*, *match_value*, and *match_label* fields, but do not specify a *d_label* field, the *data_descriptor* would be:

```
A8, IBM4341A Model4X:
```

The formatted error report would show the data as:

```
Model4X
```

If you specify *d_label*, *format*, *match_value*, and *match_label* fields, the *data_descriptor* would be:

```
PtrNode A8, IBM4341A Model4X:
```

The formatted error report would show the data as:

```
Ptrnode=Model4X
```

11.5.4.3 Error Template Example

This section shows you how an error entry using a sample error template would appear after being formatted. In the example, a hardware error entry was generated for the Serial/Parallel Adapter. The header data for the error entry includes the error ID, which in this example is 0x060103. Thus, the error identifier in the template is U13. The error entry data is listed below:

ErrorType The type of error encountered (1 ASCII character). The *format* for this data would be A1.

LastI/O The last character transmitted (1 hexadecimal digit). The *format* for this data would be X1.

LineStatus The status of the printer (1 decimal word). The *format* for this data would be D4.

PrinterStatus A 1-bit flag that indicates if the printer is active (1) or inactive (0). The *format* for this data would be B0.1.

Figure 11-13 shows a sample error template for error entries with an error ID of 0x060103. Notice the newline descriptor on the first line of the template. This will cause the data to start on a new line.

```
-------------------------------------------------------------------------

  U13 Ser/Par: \n: \
      ErrorType A1: \n: Last_I/O X1: LineStatus D4: PrinterStatus \
      B0.1:


-------------------------------------------------------------------------
```

Figure 11-13. Example of an Error Template for Error Entries with Error ID 613

Figure 11-14 shows how an error entry using the template U13 would appear in a formatted error report. The first entry is from a different error. The second entry is the one that uses template U13.

```
-------------------------------------------------------------------------

                                    ERROR LOG REPORT

  Error log: /usr/adm/ras/errfile
  ------------------------------------------------------------------------
  Date/Time            Class          Subclass       Type      Device  Cause
  Thu May  9 16:54:42  Hardware       Diskette       Counters  DSKT    Hardw
  IODN=0004       IOCN=0240
  Base_Port_Address=000003F2
  Dev_Name=DSKT
  Internal_Dev_Type=D1015200
          Switchable_to_Coprocessor
          8_bit_device
          No._Interrupts=1
          Slot_Number=1
          Adapter_Type=52
          Port_Number=00
  Bad_IO_Address_Mark
```

```
  Bad_Count=4
  Good_Count=74
  Bad_Threshold=3
  Ratio=0

  ------------------------------------------------------------------------
  Date/Time            Class           Subclass        Type      Device  Cause
  Thu May  9 16:54:45  Hardware        Ser/Par         Perm      Ser/Par Hardw
          IODN=7  IOCN=600
          Base Port Addr=0278
          Dev Name=Ser/Par
          Internal Dev Type=91002300
  ErrorType=3
  Last_I/O=44 LineStatus=0 PrinterStatus=0

  ========================================================================
```

Figure 11-14. Example Output from the Error Formatter

11.5.5 AIX Dump Facility

The AIX dump facility records the computer's state at the time of a failure. The data collected by this dump facility is intended to help you or the person servicing your system determine the cause of the failure. The AIX dump facility needs to write its output to a configured dump device/area. Therefore, before a dump can be performed, you must designate where the dump should be stored. This storage area can be either minidisk or diskette.

Subtopics

11.5.5.1 Designating a Minidisk as the Dump Area

When you install the operating system, you are given the option of creating a dump minidisk. If you did not create a minidisk during installation but want to use a minidisk for storing dumps, you can create a minidisk with the **minidisks** command.

11.5.5.2 Designating Diskette as the Dump Area

To designate diskette as the storage area for operating system dumps, do the following:

1. Edit the **/etc/system** file. Add the following line to the **sysparms** stanza:

   ```
   dump=fd0
   ```

2. Build a new kernel. At the # prompt, enter:

   ```
   newkernel
   ```

Diskette is now configured as the dump storage area.

11.5.5.3 Starting a Dump

If the system detects an operating system failure, it may start an operating system dump automatically. If the system, or part of the system, appears to be hung up (but a dump has not been started automatically), you can start an operating system dump by pressing **Ctrl-Alt-Pad 7**. If you are using a minidisk as the storage device, the dump is automatically placed on minidisk. If you are using diskette as the storage area, the system writes all of your system's real memory to diskette and requires multiple diskettes.

You should have enough formatted diskettes (approximately one for each megabyte of memory) available for your system, for example:

System Memory	Number of Diskettes
2M bytes	2
3M bytes	3
4M bytes	3

If the dump is being stored on diskette, the system prompts you to insert a formatted diskette. You may find it useful to number the diskettes in the sequence you will use them; such as 1 for the first diskette, 2 for the second diskette, and so on.

The operating system will restart automatically after a system dump.

11.5.5.4 Analyzing a Dump

To determine the cause of the system failure, use the **crash** utility to examine the operating system dump. If you are a very experienced user, you may find the **crash** command useful. See **crash** in *AIX Operating System Commands Reference* for more information.

Analyzing a Minidisk Dump Before you analyze a dump to a minidisk, you may want to save the dump. If you don't save the dump and another operating system dump occurs, the old dump on minidisk is overwritten by the new dump. To save a minidisk dump, use the **dd** command to copy the dump from the dump device (**/dev/dump**) into a file or onto diskettes. For example, to save a six-megabyte dump, you might enter the following:

```
dd if=/dev/dump of=/u/dumps/dump.xxxx bs=4k count=1536
```

where **xxxx** is a unique suffix that you use to identify that particular dump.

See **dd** in the *AIX Operating System Commands Reference* if you need more information about the **dd** command.

Analyzing a Dump to Diskette: Before you can use the **crash** command to analyze a dump on diskette, you must first use the **dd** command to copy the dump into a file. For example, to copy a three-megabyte dump into a file, you would use the following procedure:

1. Insert first diskette and enter:

   ```
   dd if=/dev/rfd0 bs=40k > /u/dumps/dump.xxxx
   ```

2. Insert second diskette and enter:

   ```
   dd if=/dev/rfd0 bs=40k >> /u/dumps/dump.xxxx
   ```

3. Insert third diskette and enter:

   ```
   dd if=/dev/rfd0 bs=40k >> /u/dumps/dump.xxxx
   ```

   where **xxxx** is a unique suffix that you use to identify that particular dump.

See **dd** in the *AIX Operating System Commands Reference* if you need more information about the **dd** command.

12.0 Chapter 12. Debugging Programs

Subtopics

12.1 CONTENTS

12.2 About This Chapter

This chapter describes the AIX language debug program. It is used to debug programs that have been compiled using the AIX C, VS Pascal or VS FORTRAN languages. You can only debug programs that are on a local machine. This chapter explains how to use **dbx** to help locate and fix program logic problems (***bugs***). Although the debug program is basically the same for the PS/2 and System/370, the sample output in this chapter uses PS/2 format.

12.3 Compiling Your Program for Debugging

The AIX compilers can optionally generate symbolic information. To generate symbolic information for an AIX C program **Cprog.c**, compile the program as follows:

```
cc -g Cprog.c -o Cprog
```

This will produce a file called **Cprog**, consisting of executable code resulting from **Cprog.c**, plus additional symbolic information generated from **Cprog.c** as a result of the **-g** option.

If an executable has more than one compilation module, each module will have its own corresponding symbolic debugging information (see the example below). If you have not compiled a particular module using the **-g** flag, that module can be debugged at the machine level only.

Example

```
cc -g Cmain.c Csub1.c Csub2.c -o Cmain
```

This will produce file **Cmain**, which will contain executable code resulting code resulting from **Cmain.c**, plus additional symbolic information generated from each of **Cmain.c**, **Csub1.c**, and **Csub2.c**.

Note: Pascal and FORTRAN programs may be compiled as in the above examples. Substitute a Pascal (**.p**) or FORTRAN (**.f**) for the C (**.c**) suffix.

12.4 The dbx Symbolic Debugger

Using the **dbx** symbolic debugger you can:

Debug programs at either the source language or assembler languag level.

Set breakpoints at selected statements and machine instructions wit conditions for activation.

Run a program one line or one instruction at a time

Access variables symbolically and display them in the correct format

Display or modify the contents of machine registers, variables, an memory.

Examine the source text using simple search functions or your ow editor.

Debug processes and subprocesses that contain **fork** and **exec** system calls.

Interrupt and examine a program that is already in progress

Trace execution of a program by line, instruction, routine, o variable.

Display expressions using a wide range of operators

Print a list (stack trace) of the active routines and thei parameters.

Print declarations of variables, along with their fully qualifie names.

Customize your interface to the debugger through command aliasing

Personalize your debugging environment with an initialization file

Call program or diagnostic routines directly from the debugger

Invoke your choice of editors and shell commands during the debuggin session.

Open a virtual terminal to separate debugger interaction from you program in HFT-supported systems.

Issue commands from either standard input or a named file

Modify the directory list from which to search for source files

Subtopics
12.4.1 Notational Conventions

12.4.1 Notational Conventions

The following notational conventions are used in this chapter:

[ ] Items enclosed in brackets are optional. Brackets enclosed in quotation marks specify actual bracket characters.

| Items separated by the *or-symbol* represent a mutually exclusive list. One and only one of these items must be chosen from the list.

In addition, items in **bold print** must be typed as shown. Information that the user supplies is printed in ***italic***.

12.5 dbx Command Summary

The following is a list of **dbx** commands with a short description of each. For more information, see *AIX Operating System Commands Reference*.

Subtopics

12.5.1 Starting dbx

dbx [*objfile* **[***corefile***]]**
Starts **dbx** using *objfile* as the program. If no **objfile** is specified, the default is **a.out**.

dbx -c *cmdfile* **[***objfile***]**
Runs the **dbx** commands in *cmdfile* before reading from standard input.

dbx -I *dir* **[***objfile* **]**
Adds *dir* to the list of directories that are searched when looking for a source file.

dbx -r *objfile*
Runs *objfile* immediately, and enters **dbx** only if the program terminates unsuccessfully.

dbx -a *pid*
Attaches **dbx** to a process already in progress with process ID = *pid*.

12.5.2 Setting and Deleting Breakpoints

stop at *linenumber* **[if** *condition***]**
Stops execution at *linenumber*.

stop in *procedure/function* **[if** *condition***]**
Stops execution when the procedure is reached.

stop *variable* **[in** *procedure/function***] [if** *condition***]**
Stops execution when the value of *variable* is modified.

stop if *condition*
Stops execution when the condition is true.

status [>*filename***]**
Displays the currently active breakpoints and trace events.

delete *event-number* **[,** *event-number, ...***]**
Removes the breakpoint and trace events associated with the event numbers.

clear *linenumber*
Removes all breakpoints at the given source line.

cleari *address*
Removes all breakpoints at the given memory location.

delete *all*
Removes all breakpoints and trace events.

12.5.3 Running Your Program from dbx

run [*args***] [<***inputfile***] [>***outputfile***] [>>***outputfile***] [2>***errfile***] [2>>***errfile***] [>&***outerrfile***] [>>&***outerrfile***]**
Starts executing program, passing ***args*** as command line arguments. Redirection of input, output, and errors occurs in the usual manner.

rerun [*args***] [<***inputfile***] [>***outputfile***] [>>***outputfile***] [2>***errfile***] [2>>***errfile***] [>&***outerrfile***] [>>&***outerrfile***]**
Same as **run**, except the default argument list is the previous argument list specified in a **run** or **rerun**.

cont [*signalnum* **|** *signalname***]**
Continues execution of the program from where it stopped. If a signal is specified, the process continues as though it received the signal.

step [*num***]**
Executes one or **num** program source line(s).

next [*num***]**
Executes up to the next or next **num** program source line(s). **next** ignores breakpoints set in called procedures.

return [*procedure***]**
Continues execution until a return to ***procedure*** is executed, or until the current procedure returns if ***procedure*** is not specified.

stepi [*num***]**
Executes one or **num** machine instruction.

nexti [*num***]**
Executes up to the next or next **num** machine instruction(s). **nexti** does not stop execution during execution of called procedures.

goto *source_linenumber*
Changes the next line to be executed to *source_linenumber*.

gotoi *address*
Changes the program counter to **address**.

12.5.4 Tracing Program Execution

trace
Prints each source line executed in the program.

trace in *procedure/function* **[if** *condition***]**
Prints each source line executed while *procedure/function* is active.

trace *source_linenumber* **[if** *condition***]**
Prints the source line when execution reaches *source_linenumber*.

trace *expression* **at** *linenumber* **[if** *condition***]**
Prints value of *expression* at *linenumber*.

trace *variable* **[in** *procedure/function***] [if** *condition***]**
Prints tracing information when the value of *variable* changes.

tracei [*variable***] [at** *address***] [if** *condition***]**
Traces one program instruction.

tracei *address* **[if** *condition***]**
Prints tracing information when the contents of *address* changes.

watch *variable* **[in** *procedure/function***] [if** *condition***]**
The same as **trace** except running under **xdbx**. Under **xdbx**, information printed is placed in a special **watch** window.

where [>*filename***]**
Displays a stack trace of procedures and parameters.

12.5.5 Ending Program Execution

quit

Exits from the debugger and the program.

12.5.6 Displaying the Source File

use *directory1 directory2 ...*
Sets the list of directories to search for source files to the directories listed in this command.

list [*linenumber_expression1* **[,** *linenumber_expression2***]]**
Displays the lines in the current source file from *linenumber_expression1* or from *linenumber_expression1* to *linenumber_expression2*. If no lines are specified, the next 10 lines are listed unless the **$listwindow** debugger variable is set to some other value.

list *procedure/function*
Displays lines surrounding the first line of *procedure* or **function**.

listi *procedure/function*
Lists instructions from the specified procedure or function.

listi at *source_linenumber*
Lists instructions beginning at the *source_linenumber*.

listi [*address1***][,** *address2***]]**
Lists instructions at *address1*, or instructions from *address1* to *address2* inclusive.

file [*filename***]**
Changes the current file to *filename*. If no file is specified, then the current source filename is displayed.

func [*procedure/function***]**
Changes the current procedure or function. If no procedure or function is specified, then the current function is displayed.

/ regular_expression **[/]**
Searches forward in the current source file for the given expression.

? *regular_expression* **[?]**
Searches backward in the current source file for the given expression.

edit [*filename***]**
Invokes an editor on *filename* or the current source file if none is specified.

edit *procedure/function*
Invokes an editor on the file containing the specified procedure or function.

12.5.7 Printing and Modifying Variables, Expressions, and Types

print *expression* **[**, *expression...***]**
Prints the values of the given expressions.

dump [*procedure***] [>***filename***]**
Prints the names and values of the variables in *procedure*, or the current procedure if none is specified.

assign *variable* **=** *expression*
Assigns the value of *expression* to *variable*.

assign *addr* **=** *expression*
Assigns the value of *expression* to location *addr* in memory.

case [default | mixed | lower | upper]
Changes the way in which symbols are interpreted. The default handling of symbols is based upon the current language. Symbols are folded to lower case unless C is the current language. This command should be used if a symbol needs to be interpreted in a way that is not consistent with the current language. Entering this command with no parameters displays the current case mode.

whatis *identifier/type*
Displays the declaration of *identifier* or *type* which may be qualified with block names as above.

which *identifier*
Displays the full qualification of *identifier* (the outer blocks that contain the identifier).

whereis *identifier*
Displays the full qualification of all symbols that match *identifier*. Order of output is not meaningful.

12.5.8 Procedure Calling

call *procedure* **(***parm1, parm2, ...***)**
Executes the object code associated with *procedure*.

print *function* **(***parm1, parm2, ...***)**
Executes the object code associated with *function* and prints any value returned by *function*.

12.5.9 Signal Handling

catch [*signalnum* **|** *signalname***]**
Causes **dbx** to trap a signal before it is sent to your program. Signals can be specified by number or by name. Signal names are not case sensitive and the SIG prefix is optional. Use this command with no arguments to list trapped signals.

ignore [*signalnum* **|** *signalname***]**
Stops trapping a signal before it is sent to your program.

12.5.10 Machine Level Debugging

address1, address2 / **[***mode***] [>***filename***]**
Prints the contents of memory starting at *address1* and continuing to *address2*. The mode specifies how memory is to be printed.

address / **[***count***] [***mode***] [>***filename***]**
Prints the contents of memory starting at **address** until *count* items have been displayed. The mode specifies how memory is to be printed.

stopi *address* | *variable* **[if** *condition***]**
Halts execution when the value of *variable* or *address* is changed.

stopi at *address* **[if** *condition***]**
Stops execution when the machine instruction is reached.

stopi if *condition* **[at** *address***]**
Stops execution when the condition is met.

tracei [at *address***] [if** *condition***]**
Traces execution of the specified machine instruction.

tracei *address* | *variable* **[if** *condition***]**
Traces modification of the specified machine location.

stepi [*num***]**
Executes one or **num** machine instruction.

nexti [*num***]**
Executes up to the next or next **num** machine instruction(s). **nexti** does not follow branch and link instructions; it continues until execution returns to the next instruction.

registers [>*filename***]**
Displays the values of the machine registers and the current instruction.

gotoi *address*
Changes the program counter to **address**.

12.5.11 Debugging Environment

alias [*aliasname* **[***commandname***]]**
Builds a user alias from the arguments specified. Specifying **alias** with no arguments displays the current aliases in effect.

alias *aliasname "commandstring"*
Builds a user alias.

alias *aliasname* **(***parm1, parm2, ...***)** *"commandstring"*
Builds a user alias from the arguments specified. Specifying **alias** with one argument displays the replacement string associated with that alias.

unalias *aliasname*
Removes an alias with the given name.

prompt "*promptstring*"
Changes the **dbx** prompt to *promptstring*.

help
Lists the **dbx** commands. Specifying the **help** command provides more information about a specific command. On an AIX/370 system, the **help** command provides a list of command names. To request more information on a specific command, type **help command**.

source *filename*
Reads **dbx** commands from *filename*.

sh *commandline*
Passes *commandline* to the shell for execution. If *commandline* is missing, the shell is entered for use until it is exited, at which time control returns to **dbx**. The **SHELL** environment variable determines which shell is used.

12.6 Invoking the dbx Debugger

There are three ways to start a debug session with **dbx**:

dbx [*options***] [***objfile* **[***corefile***]]**
This is the usual way to invoke **dbx**. If *objfile* is not specified, then **dbx** asks for the name of the object file to be examined. The default is **a.out**. If either *corefile* is specified or a file named **core** exists in the current directory, then **dbx** reports the location where the program faulted. Examining variables, registers, and memory is applicable to the core image until execution of *objfile* begins. **dbx** prompts for commands at that time.

dbx -r [*options***] [***objfile***]**
Use the **-r** option if you only want to enter the debugger if your program terminates abnormally. If *objfile* terminates successfully, **dbx** exits. Otherwise, the reason for termination is reported and you enter the debugger.

dbx [*options***]-a** *pid*
Use the **-a** option if you want to debug a process that is already in progress. In order to use this option, you must have permission to kill the process that has a process ID of *pid*. **dbx** interrupts the process if access is allowed, determines the full path name of the object file, reads in the symbolic information, and then prompts for commands. Interaction at that point is no different than if you invoked **dbx** to begin execution of the process.

The following program examples show the generation and use of an executable program called **samp**. When the program runs, it produces a ***bus error*** and stops. The commands show how to start and end the **dbx** program.

In the first example, the program **samp** has run, faulted, and caused a core image to be written. **dbx** can be used to examine the core-image dump and attempt to determine the location of the error within the program.

Compile the program with the **-g** option to get symbolic debugging capability. The symbolic information for the **samp** program will be stored in the executable file.

```
cc -g samp.c -o samp
samp
Bus error - core dumped
dbx samp
dbx version 1.1 for AIX
Type 'help' for help.
reading symbolic information ...
[using memory image in core]
    25      x[i] = 0;
(dbx) quit
```

In the second example, **dbx** can be used to examine the state of the process in memory even though a core-image dump is not taken:

```
dbx -r samp
Entering debugger ...
dbx version 1.1 for AIX
Type 'help' for help.
reading symbolic information ...
```

```
bus error in main at line 25
    25      x[i] = 0;
(dbx) quit
```

The third way of invoking **dbx** is used in order to attach to a process that is already running. Consider the following program, **looper.c**, which has a misplaced semicolon:

```
for (i = 0; i < 10; );
   printf(x[i]);
```

Compile the program with the **-g** option to get symbolic debugging capability:

```
$ cc -g looper.c -o looper
```

Run the following program from the command line:

```
$ looper
```

Seeing that your program is not terminating as expected, you want to debug it. In order to attach to **looper**, you must determine the ID number associated with the process. You must have another virtual terminal open if you did not run **looper** as a background process. From the other virtual terminal, run the system command **ps -u** *userid*, where *userid* is your login ID. All active processes that belong to you are displayed:

```
PID      TTY  TIME   COMMAND
68   console  0:04   sh
467    hft/3 10:48   looper
```

The process ID associated with **looper** is 467. Attach **dbx** to **looper**:

```
$ dbx -a 467
Waiting to attach to process 467 ...
Determining program name ...
Successfully attached to /u/user/looper ...
dbx version 2.2 for AIX
Type 'help' for help.
reading symbolic information ...
(dbx)
```

You can now query and debug the process as if it had been originally started with **dbx**.

12.7 Other Invocation Options

dbx -I *dir1* **-I** *dir2 ...* **[***options***] [***objfile* **[***corefile***]]**

Adds *dir1* and *dir2* to the list of directories that are searched when looking for a source file. Normally **dbx** looks for source files in the current directory and in the directory where *objfile* is located. If your source is in **/u/user/src** and **/u/group/src**, and *objfile* is in **/u/user/bin**, then the **-I** option should be used so that **dbx** can find the source automatically:

```
dbx -I /u/user/src -I /u/group/src objfile
```

The **use** command can be used for this purpose once **dbx** is invoked. The **use** command differs in that it resets the list of directories to be searched, whereas the **-I** option adds a directory to the list.

dbx -c *cmdfile* **[***options***] [***objfile* **[***corefile***]]**

Executes the **dbx** commands in *cmdfile* before accepting commands from standard input. The **source** command can be used for this purpose once **dbx** is invoked.

12.8 Controlling Program Execution

dbx allows you to set breakpoints (stopping places) in the program. After entering **dbx**, you can specify what lines or addresses are to be breakpoints, and then run the program with **dbx**. When the program reaches a breakpoint, it halts and reports that it has reached a breakpoint. You can then use **dbx** commands to examine the state of your program.

An alternative to setting breakpoints is to run your program one line or instruction at a time, a procedure known as ***single stepping***. In the following sections, we show how to set and delete breakpoints, begin program execution, and control program execution.

12.9 Setting and Deleting Breakpoints

Use the **stop** command to set breakpoints in **dbx**. There are four variations of the **stop** command for programs compiled with the debug flag (usually **-g**):

stop at *linenumber* **[if** *condition***]**
: A source *linenumber* is an optional file name and a : (colon) followed by a line number. For example, **(dbx) stop at "hello.c":23** or **(dbx) stop at 23**. The optional **if** *condition* specifies that execution should be halted at the specified line number if the condition is true when the line number is reached. Line numbers are relative to the beginning of the source file. A condition is an expression that evaluates to true or false. Valid expressions are described further in "Printing Variables and Expressions" in topic 12.16.

stop in *procedure/function* **[if** *condition***]**
: Use this version of **stop** to stop the program at the first executable line number in a procedure or function. For example, using C, **(dbx) stop in main** or using FORTRAN and Pascal, **(dbx) stop in programname**.

stop *variable* **[if** *condition***]**
: Use this version of **stop** to stop the program when the value of *variable* changes. For example, **(dbx) stop x**.

stop if *condition*
: Use this version to stop the program whenever *condition* evaluates to true. For example, **(dbx) stop if (x > y) and (x < 20000)**.

After any of the above commands, **dbx** responds with a message reporting the event it has built as a result of your command. The message includes the event ID associated with your breakpoint along with an interpretation of your command. The syntax of the interpretation might not be exactly the same as your command. The following are examples:

```
(dbx) stop in main
[1] stop in main
(dbx) stop at 19 if x == 3
[2] if x = 3 { stop } at "hello.c":19
```

The numbers in the brackets are the event identifiers associated with the breakpoints. When the program is halted as the result of one of the events, the event identifier is displayed along with the current line in order to show what event caused the program to stop. The events you create exist with internal events created by **dbx**, so event identifiers might not always be sequential.

Use the **status** command to display all of the current events. You can redirect output from **status** to a file. Each event is displayed in the same form as when it was generated.

There are two commands used to delete breakpoints: **delete** and **clear**. Use **delete** to eliminate events by event identifier. Use **clear** to delete breakpoints by line number. Use **delete all** to remove all breakpoints and trace events. The following examples show how to display the active events and remove them:

```
(dbx) status
[1] stop in main
[2] if x = 3 { stop } at "hello.c":19
```

```
(dbx) delete 1

(dbx) status
[2] if x = 3 { stop } at "hello.c":19
(dbx) clear 19
(dbx) status
(dbx)
```

12.10 Running Your Program

The **run** command starts your program. The command format is:

run [*args***] [<** *inputfile***] [>** *outputfile***] [>>** *outputfile***] [2>** *errfile***] [2>>** *errfile***] [>&** *outerrfile***] [>>&** *outerrfile***]**

This tells **dbx** to begin running *objfile*, passing *args* just as if they were typed on the shell command line. Input can be redirected from *inputfile* and to *outputfile* by using the preceding angle-bracket notation. The **rerun** command has the same form as **run**; the difference is that if no arguments are passed, the argument list from the previous execution is used.

Note: **dbx** may produce unexpected results if the program being run forks a child.

After your program begins, it continues until one of the following:

The program reaches a breakpoint

A signal occurs that is not ignored. For example, **INTERRUPT** or **QUIT**.

A multiprocess event occurs while multiprocess Debugging is enabled

The program completes

In each case, **dbx** receives control and displays a message explaining why the program stopped.

There are several ways to continue the program once it is stopped:

cont [*signalnum* **|** *signalname***]**
Continues the program from where it stopped. If a signal is specified, the process continues as though it received the signal. Signals can be specified by number, name, or name without the **SIG** prefix. Signal names can be either lower or upper case. The following commands are equivalent:

```
cont 3
cont SIGQUIT
cont quit
```

step [*num***]**
Runs one or *num* source line(s).

next [*num***]**
Runs up to the next or next *num* source line(s).

A common method of debugging is to step through your program one line at a time. The **step** and **next** commands are for that purpose. The distinction between these two commands is apparent only when the next source line to be run involves a call to a function or procedure. In this case, the **step** command stops in the subroutine, whereas the **next** command runs until the subroutine has finished and then stops at the next instruction after the call.

With **step**, the debugger stops after each machine instruction to see if the program has reached any line numbers. With **next**, the debugger sets an internal breakpoint at the address associated with the next line number and runs until that breakpoint is reached.

There is no event number associated with these stops because there is no permanent event associated with stopping a program.

return [*procedure***]**
Continues execution until a return to *procedure* is encountered, or until the current procedure returns if *procedure* is not specified.

If you accidentally step into a subroutine that you do not want to step through, you can use the **return** command to run through the current procedure or to a specified procedure.

12.11 Separating dbx Output From Program Output

This command is useful for debugging programs that are screen oriented, such as text editors or graphics programs. If **screen** is not used, **dbx** output is intermixed with program output.

screen

Opens a virtual terminal for **dbx** command interaction. The program continues to operate in the window in which it originated. This applies to HFT-supported systems only.

12.12 Tracing Execution

The **trace** command tells **dbx** to print information about the state of the program while the program is running. **trace** can slow your program considerably, depending on how much work **dbx** has to do. There are five basic forms of program tracing:

trace
: Single steps the program, printing out each source line that is executed. This can be very slow for the same reasons that **step** can be slow.

trace in *procedure/function* **[if** *condition***]**
: Restricts the printing of source lines to when the specified *procedure* or *function* is active. You can specify an optional *condition* to control when trace information should be produced. For example:

```
(dbx) trace in sub2
[1] trace in sub2
(dbx) run
trace in hellosub.c:    8   printf("%s",s);
trace in hellosub.c:    9   i = '5';
trace in hellosub.c:   10   }
```

trace *procedure/function* **[in** *procedure/function***] [if** *condition***]**
: Displays a message each time *procedure* or *function* is called or returned from. When *procedure* or *function* is called, the information includes passed parameters and the name or function of the calling routine. On a return, the information includes the return value from *procedure* or *function*. The following is an example:

```
(dbx) trace sub
[1] trace sub
(dbx) run
calling sub(s = "hello", a = -1, k = delete) from function main
returning "hello" from sub
```

trace *linenumber* **[if** *condition***]**
: Prints *linenumber* when the program reaches that line.

trace *expression* **at** *linenumber* **[if** *condition***]**
: Prints the value of *expression* when the program reaches *linenumber*. The line number and file are printed, but the source line is not. For example:

```
(dbx) trace x*17 at "hellosub.c":8 if (x > 0)
[1] if x > 0 { trace x*17 } at "hellosub.c":8
(dbx) run
at line 8 in file "hellosub.c": x*17 = 51
```

trace *variable* **[in** *procedure/function***] [if** *condition***]**
: Prints the location in the program and the value of *variable* each time the memory location associated with *variable* is modified. This is the slowest form of trace. The following is an example:

```
(dbx) trace x
[1] trace x
initially (at line 4 in "hello.c"):   x = 0
after line 17 in "hello.c":   x = 3
```

When a trace command is executed, the event ID associated is displayed along with the internal representation of the event. Deleting trace events is the same as deleting stop events.

12.13 Signal Handling

dbx can either trap or ignore signals before they are sent to your program. Each time your program is to receive a signal, **dbx** is notified. If the signal is to be ignored, it is passed to your program; otherwise, **dbx** stops the program and notifies you that a signal has been trapped. Change the default handling with the **catch** and **ignore** commands:

catch [*signalnum* **|** *signalname***]**

ignore [*signalnum* **|** *signalname***]**
Starts or stops trapping a signal before it is sent to your program. You can specify the signal by number or by name; names are not case sensitive, and the SIG prefix is optional. By default, all signals are trapped except **SIGCONT**, **SIGCLD**, **SIGALRM**, and **SIGKILL**. If no arguments are given, the signals currently caught or ignored are listed.

In the following example, a program uses **SIGGRANT** and **SIGRETRACT** to handle allocation of resources. You do not want to stop **dbx** each time one of these signals is received:

```
(dbx) ignore SIGGRANT
(dbx) ignore SIGRETRACT
%(dbx) ignore
CONT CLD ALARM KILL GRANT RETRACT
```

Note: **dbx** reserves the **SIGTRAP** signal for controlling the execution of the program. Therefore, abnormal behavior may result if the program receives the **SIGTRAP** signal from a process other than **dbx**.

12.14 Calling Procedures

You can call any of your program procedures from **dbx** in order to test different arguments or you can call diagnostic routines that format data in order to aid in debugging. There are two ways to call a procedure:

call *procedure* **(***parm1, parm2, ...***)**
Runs the object code associated with *procedure*.

print *procedure* **(***parm1, parm2, ...***)**
Runs the object code associated with *procedure* and prints the value returned by *procedure*.

Note: A function can be used in any expression.

12.15 Displaying a Stack Trace

To get a listing of the procedure calls preceding a program halt, use the **where** command:

where [> *filename***]**

In the following example, the executable object file **hello** consists of two source files and three procedures, including the standard procedure **main**. The program stopped at a breakpoint in procedure **sub2**.

```
(dbx) run
[1] stopped in sub2 at line 4 in file "hellosub.c"
4 printf ("%s",s);
(dbx) where
sub2(s = "hello", n = 52), line 4 in "hellosub.c"
sub(s = "hello", a = -1, k = delete), line 31 in "hello.c"
main(0x0, 0x0, 0x0), line 19 in "hello.c"
```

The stack trace shows the calls in reverse order. Starting at the bottom, the following events occurred:

1. Shell called **main** with three arguments, even though no arguments were declared in **main**. These three arguments represent the standard arguments passed into the main program: *argc, argv,* and *environ*.

2. **main** called procedure **sub** at line 19 with values s = "hello", a = -1, and k = delete.

3. **sub** called procedure **sub2** at line 31 with values s = "hello" and n = 52.

4. The program stopped in procedure **sub2** at line 4.

If the debugger variable **$noargs** is set, then the arguments passed to procedures are not displayed.

You can also display portions of the stack with the **up** and **down** commands:

up [*count***]**

down [*count***]**
Moves the current context in the stack by *count* levels.

For example:

```
(dbx) up 0
sub2(s = "hello", n = 54), line 4 in "hellosub.c"
(dbx) up 2
main(0x0, 0x0, 0x0), line 19 in "hello.c"
(dbx) down
sub(s = "hello", a = -1, k = delete), line 31 in "hello.c"
```

Refer to the appropriate chapter on Mixing Languages in the C, Pascal, and FORTRAN User Guides for a description of the system calling conventions.

12.16 Printing Variables and Expressions

You can use **dbx** to display a wide range of expressions. Specify expressions with a common subset of C and Pascal syntax, with some FORTRAN extensions.

Indirection or pointer dereferencing can be denoted using either a prefix * (asterisk) or a postfix ^ (caret), and array expressions can be subscripted by either [ ] (brackets) or ( ) (parentheses). The field reference operator for structures . (period) can be used with pointers as well as structures, making the C operator (arrow) unnecessary, although it is allowed. The & (ampersand) operator can be used to get the address of a variable.

12.17 Displaying and Modifying Variables

print *expression* **[,** *expression ...***]**
Prints the values of the given expressions.

dump [*procedure***] [>** *filename***]**
Prints the names and values of the variables in *procedure*, or the current procedure if *procedure* is not specified.

dump
Prints all live variables.

assign *variable* **=** *expression*
Assigns the value of *expression* to *variable*.

To display an expression, use **print**. To print the names and values of variables, use **dump**. If the given procedure is . (period), then all active variables are printed. To modify the value of a variable, use **assign**.

For example, in a C program, you have an automatic integer variable **x** with value 7, and you are in procedure **sub2** with parameters **s** and **n**:

```
(dbx) print x, n
7 52
(dbx) assign x = 3*x
(dbx) print x
21
(dbx) dump
sub2(s = "hello", n = 52)
x = 21
```

12.18 Scoping of Names

Names are resolved first by using the static scope of the current function, and then by using the dynamic scope if the name is not defined in the first scope. If static and dynamic searches do not yield a result, an arbitrary symbol by the same name is chosen and the message [**using** *qualified name*] is printed. The name resolution procedure can be overridden by qualifying an identifier with a block name (such as *module.variable*). In C, source files are treated as modules named by the file name without the suffix. For example, variable **x** which is declared in procedure **sub** inside file **hello.c** has a fully qualified name of **hello.sub.x**. The program itself has the name . (period). In Pascal and FORTRAN, module is the program name. For example, var x, which is declared in proc sub in program pmain inside file hello.p has a fully qualified name of "pmain.sub.x."

Two **dbx** commands are helpful in determining which symbol is found when there are multiple symbols with the same name:

which *identifier*
: Displays the full qualification of **identifier** (the outer blocks that contain the identifier).

whereis *identifier*
: Displays the full qualification of all symbols that match **identifier**. Order is not meaningful.

The following is an example after stopping in **sub2**:

```
(dbx) which s
hellosub.sub2.s
(dbx) whereis s
hellosub.sub2.s hello.sub.s hello.main.s
```

The example shows there are three procedures in the program that have a symbol named **s**.

12.19 Variables in Unnamed Blocks

In the C language, it is possible to declare variables which are "local" to a set of enclosing braces ({ }) or "unnamed blocks." **dbx** names each unnamed block uniquely within each module. For example, the first unnamed block in the module will be identified as "$b2," the second as "$b3," etc.. Variables declared within each such block are local to that block.

12.20 Expressions

An expression can be either:

A variabl

One or more operands combined with one or more operators

12.21 Operators Allowed in Expressions

The following types of operations are valid in expressions:

Algebraic: +,-,*,/(floating division), div(integral division), mod, exp(exponentiation)

Bitwise: |, bitand, xor, ~, <<, >>

Logical: or, and, not

Comparison: <, >, <=, >=, <> *or* !=, = *or* ==

Logical and comparison expressions are allowed as conditions in **stop** and **trace** commands.

12.22 Type Checking in Expressions

Expression types are checked, and can be overridden with a type renaming or casting operator. There are two forms of type renaming:

typename **(***expression***)**

expression \ *typename*

Both the typename and expression must be one of the following type specifiers:

```
in
signed in
unsigned in
lon
signed lon
unsigned lon
cha
signed cha
unsigned cha
shor
signed shor
unsigned shor
floa
doubl
long double
```

The following is an example where **x** is an integer with value 97:

```
(dbx) print x
97
(dbx) print char (x), x \ char, x
'a' 'a' 97
```

The **whatis** command prints the declaration of an identifier, which can then be qualified with block names as in the following example:

```
(dbx) whatis sub2
int sub2(s,n)
char *s;
int n;
(dbx) whatis hello.sub.k
enum status k;
```

You can also print the declaration of a c enumeration, structure, or union tag. the construct **$$***tagname* is used for that purpose:

```
(dbx) whatis $$status
enum $$status { run, create, delete, suspend };
```

The expression in an **assign** command must match the variable to which it is being assigned. If the types do not match, then an error message is displayed. The expression type can then be altered with a type renaming, or type checking can be disabled by setting a special **dbx** debugger variable named **$unsafeassign**. for example, using **n** and **status** as in the previous example:

```
(dbx) assign n = delete
incompatible types
(dbx) assign n = int (delete)
(dbx) print n, $$status (n)
2 delete
(dbx) set $unsafeassign
(dbx) assign n = suspend; print n
3
```

12.23 Folding Variables to Lower and Upper Case

By default, **dbx** folds symbols based upon the current language. If the current language is C or undefined, then the symbols are not folded; otherwise, the symbols are folded to lowercase. The current language is undefined if the program is in a section of code that has not been compiled with the debug flag. You can override default handling with the **case** command:

case [default | mixed | lower | upper]
Changes the way in which symbols are interpreted.

Using **case** without arguments tells you what the current case handling is. In the following example, the current file is **testfort.f**, a FORTRAN program:

```
(dbx) case
Symbols are folded based upon current language
Symbols are currently folded to lower case
(dbx) print X
2
(dbx) case mixed
(dbx) print X
"X" is not defined
```

The FORTRAN and Pascal compilers fold all program symbols to lowercase, whereas the C compiler does not.

12.24 Special Debugger Variables to Change Print Output

You can set the following special debugger variables in order to get different results from a **print** command:

$hexints	Prints integer expressions in hexadecimal.
$hexchars	Prints character expressions in hexadecimal.
$hexstrings	Prints the address of the character string, not the string itself.
$octints	Prints integer expressions in octal.

You can set and unset these variables to get the desired results, for example:

```
(dbx) whatis x; whatis i; whatis s
int x;
char i;
char *s;
(dbx) print x, i, s
375 'c' "hello"
(dbx) set $hexints; set $hexchars; set $hexstrings
(dbx) print x, i, s
ox177 0x63 0x3fffe460
(dbx) unset $hexchars; set $octints
(dbx) print x, i
0567 'c'
```

12.25 Displaying and Manipulating the Source File

You can use **dbx** to search through and display portions of the source files for a program. You do not need a current source listing; **dbx** keeps track of the current file, current procedure, and current line. If a core file exists, the current line and current file are initially set to the line and file containing the source statement where the process ended, unless the process was stopped in a location that was not compiled for debugging. Otherwise, they are set to the first line in main routine. While you manipulate and list the source file, the values for current line and procedure may change.

12.26 Changing the Source Directory Path

dbx normally searches the current directory and the directory where the program is located for the source files for the program. You can change this with the **-I** option on the **dbx** invocation line, or with the **use** command within **dbx**. For more information about the **-I** option, see "Other Invocation Options" in topic 12.7.

use *directory1 directory2 ...*
Sets the list of directories to search for source files to the directories listed in this command. If no directories are listed, then the current source directory list is displayed.

12.27 Displaying the Current File

The **list** command allows you to list source lines:

list [*linenumber_expression1* **[,** *linenumber_expression2***]]**
Displays the lines in the current source file from **linenumber_expression1** to **linenumber_expression2**, inclusive. If no lines are specified, the next 10 lines are listed unless the **$listwindow** debugger variable is set to some other value.

list *procedure/function*
Displays lines surrounding the first line of *procedure*.

Some special symbols representing source line numbers are useful with the **list** command, as well as with the **stop** and **trace** commands. These symbols are **$** and @. **$** represents the next line to be run, and @ represents the next line to be listed. Simple integer expressions involving addition and subtraction can be used in source line number expressions as well. For example:

```
(dbx) list $
4 {
(dbx) list 5
5 char i = '4';
(dbx) list sub
23 char *sub(s,a,k)
24 int a;
25 enum status k;
...
(dbx list @ -2
23 char *sub(s,a,k)
```

12.28 Changing the Current File or Procedure

You can use the following **func** and **file** commands to change the current file, current procedure, and current line within **dbx** without having to run any part of your program:

file [*filename***]**
: Changes the current file to *filename*. If no file is specified, the current source file name is displayed.

func [*procedure/function***]**
: Changes the current function. If no procedure or function is specified, then the current function is displayed. Changing the current function implicitly changes the current source file to the one that contains the function; consequently, it also changes the current scope used for name resolution.

You can search through the current file for text matching regular expressions. If a match is found, the current line is set to the line containing the matching text. The syntax of the search command is:

***/ regularexpression* [/]**
: Searches forward in the current source file for the given expression.

? *regularexpression* [?]
: Searches backward in the current source file for the given expression.

If you repeat the search without arguments, the search wraps around the end of a file. For example:

```
(dbx) func; file
sub2 hellosub.c
(dbx) func sub
(dbx) func; file
sub hello.c
(dbx) / i
5 static int x;
(dbx) /
6 int i = 0xfffffff;
(dbx) ? static
5 static int x;
```

You can also invoke an external text editor for your source. The default editor invoked is **vi**. You can override this default by setting the environment variable **EDITOR** to your desired editor before invoking **dbx**.

edit [*filename***]**
: Invokes an editor on *filename*, or the current source file if *filename* is not specified.

edit ***procedure/function***
: Invokes an editor on the file containing the specified procedure or function.

Control returns to **dbx** after the return from an editing session.

Subtopics
12.28.1 Debugging Programs Involving Multiple Processes

12.28.1 Debugging Programs Involving Multiple Processes

Programs involving multiple processes are those that call the system routines **fork** and **exec**. When a program forks, the operating system creates another process that has the same image as the original; the original being the parent process, and the created being the child process.

When a process performs an **exec** system call, a new program takes over or ***overlays*** the original process. Under normal circumstances, the debugger is only able to debug the original or parent process; however, **dbx** can follow the execution and debug the new processes.

You must issue the following command to enable multiprocess debugging:

multproc [on | off]
: Enables or disables multiprocess debugging. The initial value is **off**. If you issue the command with no arguments, the current status is displayed.

When multiprocess debugging is enabled and a fork occurs, both the parent and the child process are halted in the fork. A separate virtual terminal is opened for a new version of **dbx** to control the running of the child:

```
(dbx) multproc
multi-process debugging is enabled
(dbx) run
```

When the fork occurs, execution is stopped in the parent, and **dbx** displays the state of the program:

```
application forked, child pid = 422, process stopped, awaiting input

trace trap in fork at 0x1000025a (fork+0xe)
(dbx)
```

Also, another virtual terminal is opened to debug the child:

```
debugging child, pid=422, process stopped, awaiting input

trace trap in fork at 0x10000250
10000250 (fork+0x4) )c05  jcs 1000025a (fork+0xe)
(dbx)
```

At this point there are two distinct debugging sessions. The debugging session for the child retains all of the breakpoints from the parent process, but only the parent process can be rerun.

When a program performs an **exec** system call in multiprocess debugging mode, the program overwrites itself and the original symbol information becomes obsolete. All breakpoints are deleted when the **exec** occurs, and the new program is stopped and identified in order for the debugging to be meaningful. **dbx** attaches itself to the new program image, makes a system call to determine the name of the new program, reports the name, and then prompts for input:

```
(dbx) multproc
Multi-process debugging is enabled
(dbx) run
Attaching to program from exec...
```

```
Determining program name...
Successfully attached to /u/user/execprog...
Reading symbolic information...
(dbx)
```

Notes:

1. It can take a fairly long time for **dbx** to determine the name of the overlaying program.

2. It is not possible to determine the name of a program that belongs to a remotely-mounted file system.

12.29 The dbx Debugging Environment

The following topics show how to modify and change the **dbx** environment.

12.30 The Alias Facility

You can build your own commands from the **dbx** primitive command set. The following commands allow you to build a user alias from the arguments specified. All commands involved in the replacement string for the alias must be **dbx** primitive commands. You can then use your aliases in place of the **dbx** primitives.

alias [*aliasname* **[***commandname***]]**

alias *aliasname "command string"*

alias *aliasname* **(***parm1, parm2, ...***)** *"command string"*

The **alias** command with no arguments displays the current aliases in effect; with one argument the command displays the replacement string associated with that alias.

The first two forms of **alias** are used to simply substitute the replacement string for the **alias** each time it is used. For example:

```
(dbx) alias rr rerun
(dbx) alias step2 "step;step"
```

Each time **rr** is typed at the command line, **dbx** performs a **rerun** command. Similarly, **step2** results in two **step** commands being executed.

The third form of aliasing is a limited macro facility. Each parameter specified in the **alias** command is substituted for in the replacement string. This can be useful in eliminating excessive typing:

```
(dbx) alias px(n) "set $hexints; print n; unset $hexints"
(dbx) alias a(x,y) "print symname[x] symvalue._n_n.name.Id[y]"
(dbx) px(126)
0xfe
```

The alias **px** in the previous example prints a value in hexadecimal without permanently affecting the debugging environment. The following aliases and associated command names are supplied by **dbx** by default:

t	**where**	**p**	**print**
j	**status**	**n**	**next**
st	**stop**	**l**	**list**
s	**step**	**e**	**edit**
r	**run**	**h**	**help**
x	**registers**	**d**	**delete**
q	**quit**	**c**	**cont**

You can remove an alias with the **unalias** command:

unalias *aliasname*
Removes an alias with the given name.

12.31 Changing the dbx Prompt

The **dbx** prompt is normally the name with which you invoked **dbx**. If you specified **/usr/bin/dbx a.out** on the command line, then your prompt would be **/usr/bin/dbx**. You can change the prompt to be whatever you desire with the **prompt** command:

prompt *"promptstring"*
Changes the **dbx** prompt to *promptstring*.

For example:

```
/usr/bin/dbx hello
dbx version 2.2 for AIX
Type 'help' for help.
reading symbolic information...
(/usr/bin/dbx) prompt "dbg>"
dbg>
```

12.32 Customizing Your Environment With .dbxinit

Each time you begin a debugging session, **dbx** searches for a special initialization file named **.dbxinit**. This file is searched for first in the current directory, and then in your home directory. **.dbxinit** should contain a list of **dbx** commands to execute each time you begin a debugging session. These commands are executed before **dbx** begins to read commands from standard input. Normally, **.dbxinit** contains **alias** commands, but it can contain any valid **dbx** commands. For example:

```
cat .dbxinit
alias si "stop in"
prompt "dbg>"
dbx a.out
dbx version 2.2 for AIX
Type 'help' for help.
reading symbolic information...
dbg> alias
si      stop in
t       where
...
dbg>
```

12.33 Reading dbx Commands From a File

The **-c** invocation option and **.dbxinit** provide mechanisms for executing **dbx** commands before reading from standard input. There is also a way to read **dbx** commands from a file once the debugging session has begun:

source *filename*
Reads **dbx** commands from the given *filename*.

12.34 Running Shell Commands From dbx

You can run shell commands without exiting from the debugger:

sh *commandline*
Passes *commandline* to the shell for execution.

If *commandline* is missing, the shell is entered for use until it is exited, at which time control returns to **dbx**. The **SHELL** environment variable determines which shell is used. For example:

```
(dbx) sh echo $SHELL
/bin/sh
(dbx) sh
echo 'This is the shell' #You will remain in the shell until you exit.
exit
(dbx)
```

12.35 Getting Help

help
Prints a synopsis of common **dbx** commands.

12.36 References

The **cc** command in *AIX Operating System Commands Reference* and the compiler chapters in *AIX VS Pascal, VS FORTRAN, and C User's Guides*.

12.37 dbx Vector Processor Support

dbx vector processor support is an enhancement to **dbx** which now supports 370 Fortran. This enhancement consists of a number of new commands which are entered at the (**dbx**) prompt. One type of command displays the contents of a specified vector register or all vector registers. The contents of individual registers may be displayed in binary, floating point, or double precision format. In addition, the contents of any of the registers of the vector processor can be modified.

dbx contains a number of internal debugger variables which can be set by the user from the command line. New debugger variables have been added to support the vector processor enhancements. In general, these new variables either control the default display format of certain commands or refer to the vector processor registers themselves.

scalar
: A single data item, which may be floating point number, a binary integer, or logical data.

vector
: A linearly ordered collection of scalars.

Element
: Any single scalar component of a vector, regardless of data type.

section size
: The number of locations in a vector register; always a power of two, where 8 <= SS <= 512.

The format and operation of the **dbx** vector processor commands is explained in the following sections.

Subtopics

12.37.1 dbx Vector Processor Support Commands

The following commands have been added for **dbx** vector processor support. These commands only apply to AIX/370.

vregisters [<vreg number>]

Prints the contents of all vector registers or the register designated by **vreg number**. The format of the display is controlled by internal debugger variables as follows:

If **$intvectors** debugger variable is set, the contents of the vector elements are displayed as integers.

If **$doublevectors** debugger variable is set, the contents of the vector elements are displayed as double precision floating point numbers. When specifying the vregisters with a single register, the **vreg number** parameter must be even.

The above two variables are mutually exclusive. If one is set, the other is automatically cleared.

vstate

Prints the contents of the vector status register, vector mask register, and vector activity-count register.

12.37.2 Debugger Variables

The variables below are special variables known to the debugger. They may be displayed with the **print** command or modified with the **assign** command.

$vrN (where 0 = N = 31)

References vector register N as an integer. The reference must be further qualified to specify a particular component of the register. For example, $vr5[13] references the 14th word of the 6th vector register (vector registers and components within registers are numbered starting with 0).

$vfrN (where 0 = N = 31)

References vector register N as a single precision floating point number. A particular component within the register must be specified.

$vdrN (where 0 = N = 30, n even)

References vector register N as a double precision floating point number. A particular component within the register must be specified.

$vstatus References the vector status register. The display appears as two 32-bit words in hexadecimal format. The two halves of the register may be referenced separately using the subscript notation.

$vac References the vector activity-count register. The display appears as two 32-bit words in hexadecimal format. The two halves of the register may be referenced separately using the subscript notation.

$vmask References the vector mask register. The display appears as a number of 32-bit words displayed in hexadecimal format. Individual words may be referenced separately using the subscript notation. The number of words displayed depends on the current section size. For example, if the section size is 128, four 32-bit words will be displayed to show 128 bits, one bit per vector element.

$vcount Displays the vector count field of the vector status register in decimal. This field is a 16-bit unsigned binary integer.

$vmaskflag Displays the vector mask mode bit of the vector status register.

$vstate Controls whether status information from the vector processor will be displayed as part of the output of the **registers** command. If **$vstate** is set, the contents of the vector status register, the vector activity-count register, and the vector mask register will be displayed in hexadecimal in addition to the normal output of the **registers** command.

Note: The **registers** command should not be confused with the **vregisters** command which applies strictly to the vector processor. The **registers** command is a **dbx** command which displays the values of the general purpose registers and

system control registers.

See the AIX 370 Fortran manual on use of the vector hardware for computation purposes. For additional information on the **dbx** interface to the vector processor hardware, refer to the **ptrace** and **sysconf** system calls in the *AIX Operating System Technical Reference*. See the *IBM Enterprise Architecture/370 and System/370-Vector Operation* for additional information on the hardware.

12.37.3 Error Handling

If you attempt to use the vector commands on a 370 system which does not have the vector hardware, you receive the following message:

```
no vector support
```

If any of these commands are input to the PS/2 debugger, it responds the same way it would to any other unrecognized command.

Other diagnostic messages inform the user of incorrect formats if they attempt to use the debugger variables which refer to the hardware registers (for example, using $vr0 without a subscript).

13.0 Chapter 13. Installing and Updating an LPP

Subtopics

13.1 CONTENTS

13.2 About This Chapter

This chapter is provided for software developers who will write programs to run on the AIX platform. It describes the AIX tools used for installing and updating programs. By designing your application to use these tools, your users will be able to install and update you application by the same procedures they use for AIX Licensed Program Products (LPPs).

After finishing the development and testing of a new program, you must prepare it to be installed. This chapter describes the tools used for the installation of LPPs. The descriptions include:

What the installation services d
What files to include on the distribution medi
The format of each of the needed files

At a later time you may need to change the LPP to provide additional features, or to correct problems encountered in actual usage. This chapter also describes the tools used to change an installed LPP. The descriptions include:

What the update services d
What files to include on the update medi
The format of each of the needed files

13.3 Understanding System Guidelines

When preparing your LPP to install on the AIX system, be aware of how the system files are organized so that the LPP can work as an integral part of the whole system. The following paragraphs describe some of the system conventions that any new LPP should follow.

Subtopics
13.3.1 System Directories
13.3.2 Providing User Documentation

13.3.1 System Directories

Each of the directories supplied with the system has a defined purpose. The permission, owner and group ID for these directories are set so that the directories work properly with the system LPPs that use them.

CAUTION:
Do not change the permission, owner or group id of any system file or directory. Such changes may compromise proper operation of system programs.

Many of the administrative system functions can be performed by a member of the **system** group without having to log in as **root** or know the **root** password. Refer to the description of each command in *AIX Operating System Commands Reference* to determine which functions the **system** group can perform. Some sensitive commands are still reserved for the person with superuser authority.

In addition, reserve the following directories for their intended purpose only:

/ Do not add files or mount file systems to subdirectories in the root file system without an urgent need to do so. The root file system contains fixed system directories and should not be used for application programs or user files.

/usr Do not add files or directories to the **/usr** directory without an urgent need to do so. The **/usr** directory contains files and directories that the system uses.

/usr/bin This directory contains common user executable files and is part of the default search path for all users. Put a short binary program or shell script in this directory that calls your large executable files. Put the large executable files in the directory **/usr/lpp/***lpp-id* (*lpp-id* is the identifier you choose for your LPP, as described in "What You Need to Install an LPP" in topic 13.6).

/usr/lpp Create a directory in this directory to store your LPPs as described in "What You Need to Install an LPP" in topic 13.6.

/usr/lib Use this directory to store those library files that have general use to all users. Store other library files in your directory, **/usr/lpp/lpp-id**.

/usr/include Use this directory to store those include files that have general use to all users. Store other include files in your directory, **/usr/lpp/lpp-id**.

13.3.2 Providing User Documentation

You should provide documentation along with your LPP, describing its installation and use. For uniformity, pattern installation procedures after the format and steps shown in *Installing and Customizing the AIX Operating System*. If your LPP includes new device drivers, refer to the *AIX Operating System Technical Reference*. In addition, you should provide information about your LPP's use of disk space in each of the following directories:

/usr
/usr/bin
/usr/lib
/usr/lpp
/usr/include

Give usage information in terms of the number of 4K-byte blocks that the LPP uses in each of the directories. The usage information helps the person installing your LPP determine if there is enough space on the file system to load the LPP. If there is not enough space, refer the user to *Managing the AIX Operating Systems* for information about expanding an existing file system.

13.4 Using Installation and Update Services

The installation and update services require only a few simple steps to install the LPP on the system. Typically, whoever installs the LPP need only enter one of the following two commands:

installt options if the medium is PS/2 tape, or

installp options if the medium is anything else.

where **options** are as defined in the *AIX Operating System Commands Reference*. From that point on, messages to the screen guide the user.

In addition to providing a simple user interface, the installation and update services also help ensure that the LPPs are handled correctly. They:

Maintain an accurate record of the revision state of each LPP on th system

Verify the existence and revision level of other needed LPP

Provide instruction

Provide online copies of changes to documentation

Subtopics

13.4.1 Commands (installp, installt, updatep)

Installation and update services provide the following commands to be entered from the command line to install or update an LPP. Complete information about the syntax and usage of these commands is in *AIX Operating System Commands Reference*. Further information on the **installt** command can be found in Appendix D of this book.

Command	Description
installp	Installs LPPs.
installt	Installs LPPs or user packages from PS/2 tape.
updatep	Applies one or more changes to an installed LPP.

For simplicity, these commands will be referred to collectively as the service tools.

13.4.2 Replicated and Non-replicated File Systems

The root file system is a system replicated file system. It is found on each site in the cluster and contains files and directories that are required for all cluster sites. In order to maintain consistency of all root copies, AIX requires that one particular copy of a file system be available before any modification is made. This is called the ***primary*** copy. The cluster site that has the primary copy is called the ***primary site***. The root file system on each cluster site is a copy of the primary copy. The primary site controls all modifications to the root file system and makes sure they are propagated to all other cluster sites.

The service tools require that the primary copy of the root file system be available for changes. When service affects the replicated root file system, all other cluster sites will automatically be affected. When changes are made to a non-replicated or local file system, a different mechanism is used to propagate changes to other sites.

The special mechanism developed for local file systems uses a ***system queue***. The system queue is a set of directories that contain action information (**apply**, **commit**, **uncommit**, **reject**) for local sites. The queue is accessed through two programs, **qapp** and **qproc**. The purpose of the queue is to keep track of all **installp/updatep** changes to the local file systems. When **installp/updatep** is executed, entries are put into the system queue via **qapp** if local action is needed. When **qproc** is run on each site, it processes any local changes. This queuing system allows for synchronization of **installp/updatep** changes to local file systems in a time independent fashion, so that sites that are down or that have not joined the cluster can be brought up-to-date at a later time.

A change to the kernel is an example of a change that requires local action. The installation of an application in **/usr/bin** is an example of a change that does not require local action.

In summary, any changes to files on the root file system are automatically propagated throughout the cluster. Changes to files on the local non-replicated file systems are propagated via the system queue mechanism. Every cluster site will normally get these changes, although this can be tailored in **inst_updt.loc**.

13.4.3 installp and updatep File Hierarchy

It is important to understand the **installp** and **updatep** file system hierarchy.

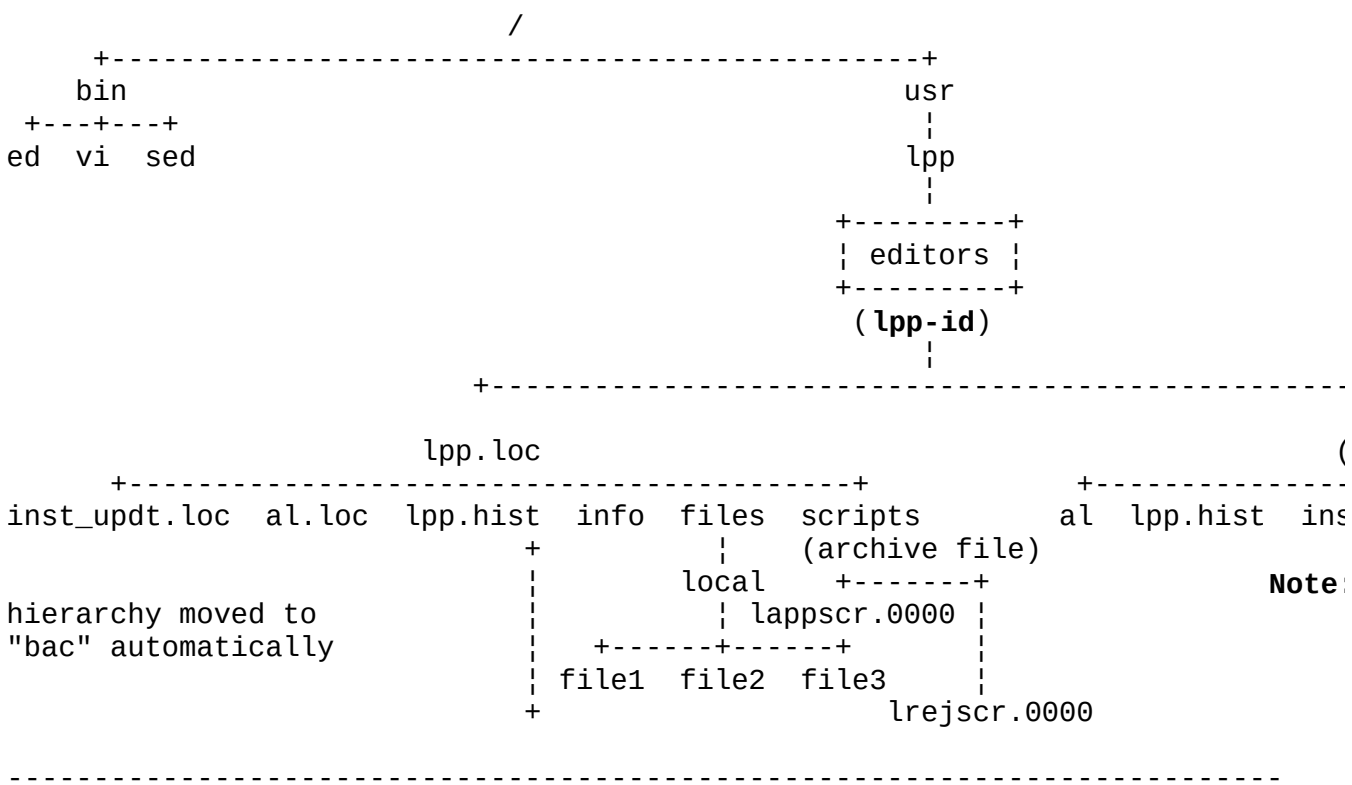
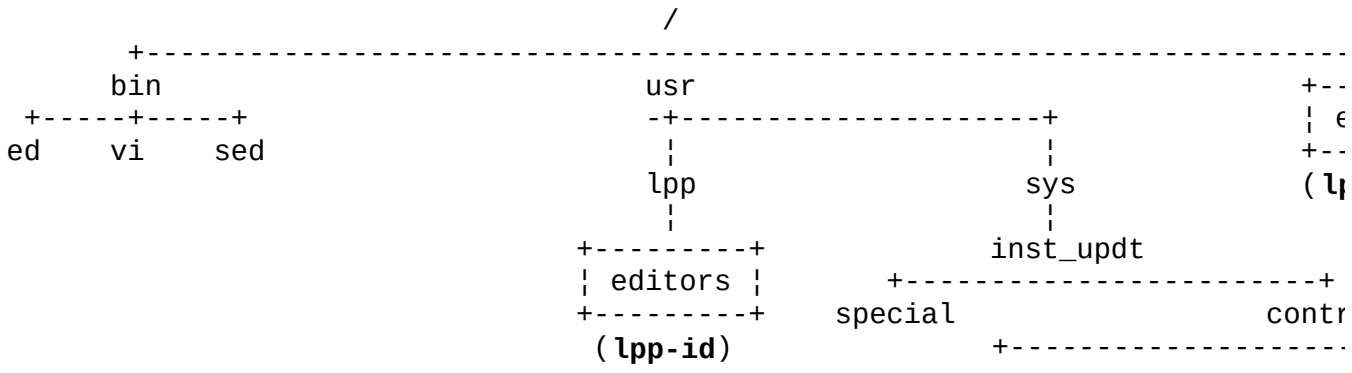


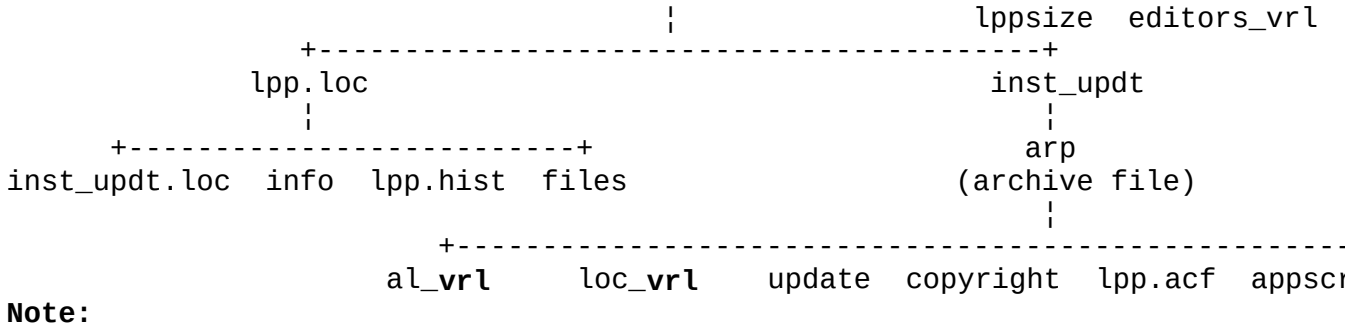
Figure 13-1. installp File Hierarchy - Fixes

This example shows an LPP called "editors" with files "ed," "vi," and "sed" in the root file system under **/bin**. Note that the local files (those that are installed on each site in the cluster) are under **/usr/lpp/***lpp-id***/lpp.loc/files/local**. At a later time or at the user's discretion, the queuing mechanism will run the vendor supplied executable file **inst_updt.loc** on each site. These local files are then copied into that site's local filesystem.

Note: "editors" is an example LPP.

Figure 13-2 shows an example file hierarchy for **updatep** media (similar to the **installp** example).





Note:
lpp-id = editors
vrl = version release level number
nnnn = level number

Figure 13-2. updatep File Hierarchy

13.4.4 Installation Files

The following is a list of all files used for an **installp** installation, apart from the actual files being installed.

Figure 13-3. installp Files

FILE	TYPE
/lpp_name	Required
/copyright	Optional
/usr/lpp/*lpp-id***/liblpp.a**	Required (archive file)
al	Required (file)
instal	Required (executable)
prereq	Prereq Optional (file)
lpp.hist	Required (file)
copyright	Optional (file)
lpp.acf	Optional (file)
/usr/lpp/*lpp-id***/lpp.loc**	Optional (directory)
al.loc	Required if **lpp.loc** exists (file).
files	Required if **lpp.loc** exists (directory).
info	Required if **lpp.loc** exists (file).
inst_updt.loc	Required if **lpp.loc** exists (executable fil).
lpp.hist	Required if **lpp.loc** exists (history file).
scripts	Optional (archive file)
lappscr.0000	Optional (executable file)
lrejscr.0000	Required if **lappscr.0000** exists (executable file).

13.4.5 Update Files

The following is a list of all files used for an **updatep** upgrade. *level-nr* indicates the level number.

Figure 13-4. updatep: Files for an upgrade

FILE	TYPE
/lpp_name	Required
/copyright	Optional
/usr/lpp/*lpp-id***/inst_updt/arp**	Required (archive file)
al_*vrl*	Required (file)
loc_*vrl*	Optional (file)
update	Required (executable)
copyright	Optional (file)
lpp.acf	Optional (file)
appscr.*level-nr*	Optional
rejscr.*level-nr*	Required if **appscr.***level-nr* exists
lappscr.*level-nr*	Optional (file)
lrejscr.*level-nr*	Required if **appscr.***level-nr* exists
/usr/lpp/*lpp-id***/lpp.loc**	Optional (directory)
files	Required if **lpp.loc** exists (direct
info	Required if **lpp.loc** exists (file)
inst_updt.loc	Required if **lpp.loc** exists (execut
lpp.hist	Required if **lpp.loc** directory exis
/usr/sys/inst_updt/control	Required (archive file)
lppsize	Required (file)
*lpp-id*_**vrl**	Required (file)
*lpp-id*_**instr**	Optional (archive file)
*lpp-id*_**erata**	Optional (archive file)
/usr/sys/inst_updt/special	Required (file)

13.4.6 installp and updatep Script Considerations

Subtopics
13.4.6.1 Executable File Considerations
13.4.6.2 installp and updatep Script Descriptions

13.4.6.1 Executable File Considerations

The service tools use certain scripts. These files **must** have their permissions set to executable. Figure 13-5 shows the script files.

Figure 13-5. Scripts Used by the Service Tools

Invoked By	Non-local File	Local File
installp	**instal**	**inst_updt.loc** **lappscr.0000** **lrejscr.0000**
updatep	**update** **appscr.nnnn** **rejscr.nnnn**	**inst_updt.loc** **lappscr.nnnn** **lrejscr.nnnn**

Note: **updatep** and **installp** can only be used by a user with superuser authority.

13.4.6.2 installp and updatep Script Descriptions

The following is a description of each script:

instal Required on all **installp** media. Used by **installp** only. This script moves files in the apply-list **al** from the media to the root filesystem. It can also edit files and use various other commands such as **mknod**, **ln**, etc.

update Required on all **updatep** media. Used by **updatep** only. This script moves files in the apply list **al** from the media to the replicated filesystem. It also takes care of saving old versions of updated files before they are overlaid.

appscr.nnnn Optional on all **updatep** media. Used by **updatep** only in the **apply** phase. This script edits files, makes links, etc., and does all the work for applying a particular level update (except for overlaying files by copying them from the media to the replicated filesystems).

rejscr.nnnn Optional on all **updatep** media. Used by **updatep** only in the **reject** phase. This script edits files, removes links, etc., and undoes the effects of the earlier run **appscr.nnnn**.

Note: If an **appscr.nnnn** exists, then the corresponding **rejscr.nnnn** is required.

inst_updt.loc Optional on all **updatep** and **installp** media. Used by **installp** and **updatep** (on local filesystems). This script moves files in the apply list **al.loc** from the queue directory to a particular site's local filesystem. It should not do any work other than copying files. Actions such as editing files, creating links, etc. are taken care of by the **lappscr.nnnn** script. The **inst_updt.loc** script may also return a code requiring the system to make a kernel or ask the user to reboot. It may also return an error code in case of a failure. Refer to "Installation and Update Procedure Return Codes" in topic 13.6.4.5.

lappscr.nnnn Optional on all **updatep** and **installp** media. Used by **installp** and **updatep** in the **apply** phase (on local filesystems). This script edits files, makes links, etc. on the local filesystem. It returns an error code in case of a failure.

lrejscr.nnnn Optional on all **updatep** and **installp** media. Used by **installp** and **updatep** in the **reject** phase (on local filesystems). This script edits files, removes links, etc. on the local filesystems and undoes the effects of the earlier run **lappscr.nnnn**. It returns an error code in case of a failure.

Note: If an **lappscr.nnnn** exists, then the corresponding **lrejscr.nnnn** is required on the same media.

13.5 Order of Execution

This section describes the order of execution for the following:

installp

updatep apply

updatep reject

Subtopics

13.5.1 installp Order of Execution

The following is the order of execution for the scripts during **installp**:

Replicated file

installp: **instal** is run.

Local file

apply: **inst_updt.loc** is run, and the return code is remembered. **lappscr.0000** is run and any action that is requested by the return code of **inst_updt.loc** is performed (for example, make a kernel). Reboot may be required.

13.5.2 updatep apply Phase Order of Execution

The following is the order of execution during the **updatep apply** phase:

Replicated file

update is run, followed by **appscr.nnnn** scripts in order from the lowest numbered **nnnn** to the highest.

Local file

inst_updt.loc is run, and the return code is remembered. Next, the **lappsrc.nnnn** scripts are run from the lowest numbered **nnnn** to the highest. Any action that is requested by the return code of **inst_updt.loc** is performed (for example, make a kernel). Reboot may be required.

13.5.3 updatep reject Phase Order of Execution

The following is the order of execution during the **updatep reject** phase:

Replicated file

The **rejscr.nnnn** scripts are run from the highest number to the lowest. Then, **/etc/inurecv** is called to recover overlaid and deleted files.

Local file

lrejscr.nnnn is run, from the highest numbered **nnnn** to the lowest. Then, **/etc/lpp/inrcvl** is called to recover overlaid and deleted files. Finally, the return code from the original **apply** step is examined and the appropriate action is taken.

13.6 What You Need to Install an LPP

The primary site must be up to install an LPP. To install an LPP on the AIX system using the **installp** command, provide a set of control and LPP files in **backup** format on the installation media. The control files must appear before the LPP files. In addition, when backing up the files, use a relative path name with respect to the **/** directory on the installed system. For example, to create installation media for a single LPP file (**ftrn**) to be stored in the **/bin** directory on the target system, provide an ordered list of files to the **backup -i** command:

```
./lpp_name
./usr/lpp/FORTRAN/liblpp.a
./bin/ftrn
```

The first two files are control files; the last file is the LPP file (or a group of files). There is no minimum required list of LPP files.

Subtopics

13.6.1 General Control Files

Provide the following control files at the beginning of the installation media:

A file named **lpp_name** that contains the title of the LPP or LPPs on the installation media (see "The LPP Name File" in topic 13.9.6). This file **must** appear first on the media.

A library file named **./usr/lpp/***lpp-id***/liblpp.a** (*lpp-id* is a name for the LPP with a maximum of 8 characters). Create this library using the **ar** command. The library contains the following files:

instal The installation procedure that installs special features and files that the LPP uses. This procedure must be either a shell script or an executable program in **a.out** format. This script may do the following:

- Copy files in **al** from LPP media to the system
- Edit files, make directories, etc..

lpp.hist A file for logging changes to the LPP (see "Creating the LPP History File" in topic 13.9.1).

al An apply list file that contains the relative path names with respect to the **/** directory of all files to be restored.

copyright An optional file that contains appropriate copyright information for this LPP.

prereq An optional file that lists other LPPs that this LPP uses and that, therefore, must be installed on the system (see "Creating the LPP Requirements File" in topic 13.9.4).

lpp.acf An optional file that defines archiving procedures for the LPP (see "The Archive Control File" in topic 13.9.8).

Other files Other optional files can be added to the end of the archive file as needed.

13.6.2 Special Files for the Local

The following files and directories are required for the local installation:

/usr/lpp/*lpp-id***/lpp.loc**

A directory containing the following files:

inst_updt.loc Executable which does the local installation.

al.loc Apply list of all files needed for local installation.

lpp.hist A history file for the local installation.

info A file containing installation information and local file system size requirements.

/usr/lpp/*lpp-id***/lpp.loc/files**

A directory containing files needed by the **inst_updt.loc** script.

Example: Assume that you have an LPP called PCI. You want to install the files PCI1, PCI2, etc. on the local filesystem. If you want to install **/local/bin/PCI1** and **/local/sys/PCI2**, you must have an apply list, called **al**, in **./usr/lpp/PCI/liblpp.a**. In addition to the usual general control files this apply list must have the following files:

./usr/lpp/PCI/lpp.loc/inst_updt.loc

./usr/lpp/PCI/lpp.loc/al.loc

./usr/lpp/PCI/lpp.loc/lpp.hist

./usr/lpp/PCI/lpp.loc/info

./usr/lpp/PCI/lpp.loc/files/local/bin/PCI1

./usr/lpp/PCI/lpp.loc/files/local/sys/PCI2

You must also have an **al.loc** that has the following entries:

./local/bin/PCI1

./local/sys/PCI2

On your media, you must have the following:

./usr/lpp/PCI/lpp.loc/files/local/bin/PCI1

./usr/lpp/PCI/lpp.loc/files/local/sys/PCI2

In addition to the above files, you must also have an **info** file that contains size and LPP information. The final file that you need to have is the local history file **./usr/lpp/PCI/lpp.loc/lpp.hist** which specifies the **commit** status and other LPP information.

For more information about the specific files, refer to "Example of Files on installp Media" in topic 13.6.5.

13.6.3 General LPP Files

The LPP files are the files needed to use the LPP. You must include these files after the control files on the media.

13.6.4 Creating the Installation Script

To install an LPP and its files, you must provide an executable file named **instal**, which is usually a shell script. This **instal** script must be contained in the archive file **./usr/lpp/***lpp-id***/liblpp.a**. The **installp** program restores the archive file **./usr/lpp/***lpp-id***/liblpp.a** from the installation media and extracts the files in it. Then, **installp** executes your **instal** script and passes it a single parameter that specifies the device containing the **restore** files.

When **instal** returns to **installp**, **installp** removes all files that do not begin with **lpp** from directory **/usr/lpp/***lpp-id*. Directories in that directory are not deleted, however. The following is an example of a simple **instal** script:

```
#     Restore all replicated files.
/etc/inurest -q -d$1 /usr/lpp/ate/al ate
exit $?
#
#     end
#
```

The following is a list of tasks that your **instal** script must perform to complete an installation:

1. Ensure that the necessary level of other LPPs you depend on are installed on the system using **ckprereq**.

2. Use **inusave** to back up any files that will be replaced or deleted.

   CAUTION:

   If your new LPP has files with names that conflict with file names in other installed LPPs, you need to add commands to detect those files and move them out of the way. **installp** will not override a file that already exists on the system. In addition, do not use **inusave** when installing large LPPs that will fill the disk with the backup files.

3. Use **inurest** to restore files listed in the apply list file, and to archive constituent files.

4. Customize the system for the LPP. See *AIX Operating System Technical Reference* for information to help you customize the system.

5. If an error occurs and **inusave** was previously used, then use **inurecv** to recover the previous state of the system.

6. If **inusave** was used, delete the directory, **/usr/lpp/***lpp-id***/inst_updt.save**.

7. Return a completion code to **installp**.

Subtopics

13.6.4.1 Special Script for the Local

When **installp** determines that there is some local installation to do, it uses **qapp** to set it up. The **qapp** program puts the files under **.../lpp.loc/files** in a backup-by-name saveset called **bac**. It creates an entry in **/usr/lib/qproc.queue**, as shown in Figure 13-6, for use by **qproc**.

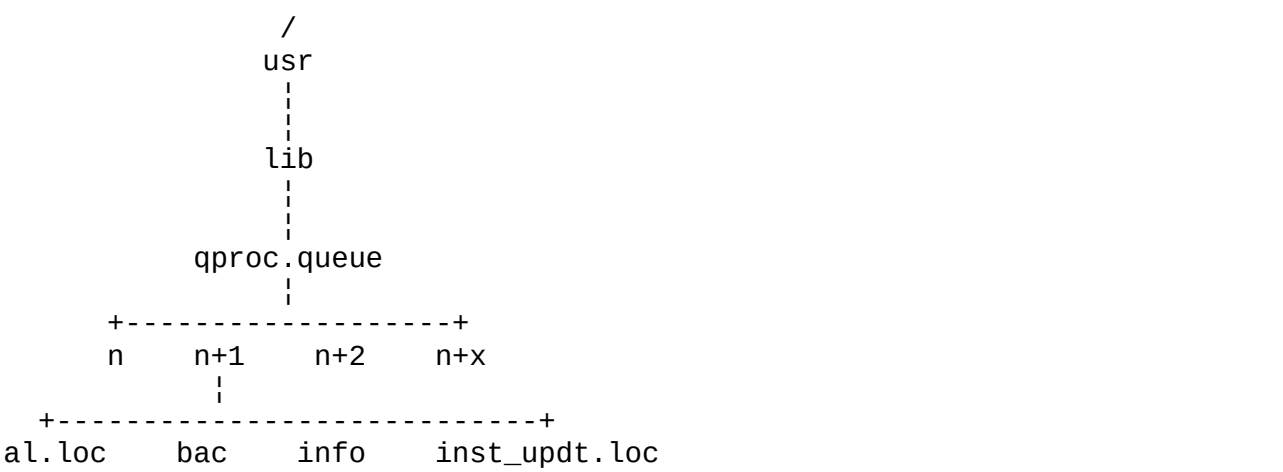


Figure 13-6. Files for a local installation/update

On each site, the **inst_updt.loc** script runs in the directory that contains the files **bac** and **al.loc**. The files in **bac** are in backup-by-name format, to be extracted using **al.loc** as the file list.

The **inst_updt.loc** script should do the following:

```
current=`pwd`
cd /
/etc/inurest -q -d$current/bac $current/al.loc lpp-id
```

In the above example, the first line remembers the current directory, which also contains **al.loc** and **bac**. The second line changes the directory to root. The third line restores the files from **bac** to their proper location on the local filesystem.

This may be more easily understood by looking at the file and directory hierarchy (see Figure 13-1 in topic 13.4.3).

Note: If the LPP is to be installed on a particular CPU type, then the script **inst_updt.loc** should use the **site** command to determine if the CPU is the desired type. If it is not the desired type, then the script should exit immediately with a zero return code.

13.6.4.2 Restoring the LPP Media

At some point in the **instal** script, **inurest** is used to restore the LPP media. For example, to restore the files listed in the apply list file **al** for the LPP named **myprog**, use a shell command in the following format:

```
/etc/inurest -qd device /usr/lpp/myprog/al myprog
```

This command restores all files from the specified device. **instal** uses the device passed to it by **installp**. The parameter **/usr/lpp/myprog/al** is the full path name of the apply list file, and **myprog** is the name of the LPP being installed.

Also use the **inurest** command to archive and then delete any restored archive member files. To enable the archive process, include a file **lpp.acf** in the archive file **./usr/lpp/***lpp-id***/liblpp.a**. See "The Archive Control File" in topic 13.9.8 for the format of this file. If this file is present, **inurest** archives and then deletes all restored files that are listed in both **lpp.acf** and the apply list file (**al** in the example).

13.6.4.3 Allowing for Individual Needs

The **instal** script may need to perform some special tasks to ensure that the LPP operates properly. These may include the following:

Use the **inurest** command together with an apply list file and archive control file to store files in a library.

Display progress messages during the different stages of th installation procedure.

Recover from errors, or back out of the installation if errors occur

13.6.4.4 Customizing the System for an LPP

Some changes require modification of configurable system files. These files are in general not overlaid, but modified through editing or commands such as **echo**. These files may be local (i.e. **dfile**) or replicated (i.e. **profile**). Such files must be changed by using the following scripts:

instal For installs affecting replicated files.

lappscr.0000 For installs affecting local files.

appscr.nnnn For updates affecting replicated files.

lappscr.nnnn For updates affecting local files.

Before a script changes such files, it is advisable to back them up to another location. The script can check if the modification failed and if so, it can move the backed-up copy of the file to its original location and exit with an error code.

Whenever one of the above scripts does its intended action, there must be a corresponding script to "undo" its effects, for use in **rejects**.

Forwards Script	**Backwards Script**
instal	
lappscr.nnnn	**lrejscr.nnnn**
appscr.nnnn	**rejscr.nnnn**

If there is a need to reboot, the local script **inst_updt.loc** should return a reboot code. See the following section for a list of the proper return values.

13.6.4.5 Installation and Update Procedure Return Codes

When an installation or update has completed, it may be necessary to rebuild the kernel or do a reboot. The **instal** updates and **inst_updt.loc** procedures may initiate the desired action by using the following return codes:

Figure 13-7. Return Codes from instal, and inst_updt.loc

Code	Description
0	Successful completion. No further actions required.
2	Successful completion. Do a **sync** (**inst_updt.loc** only) reboot and exit.
3	Successful completion. Rebuild the kernel, do a **sync** (**inst_updt.loc** only). The system will ask the user to reboot when all installation updates are completed.
4	Successful completion. Rebuild the kernel, do a **sync**, (**inst_updt.loc** only) tell the user that reboot is required and exit.
6	Successful completion. Do a **sync** (**inst_updt.loc** only). The system will ask the user to reboot when all installations record or updates are completed.
7	The **update** procedure requests cancellation of the update with errors. The service tools recover the previous state of the system (**update** only).
Other	Error. If returned from **instal**, then write the information to **lpp.hist** with zeroes in the **Version**, **Release**, **Level** fields, and the return code value as the error code field.

13.6.5 Example of Files on installp Media

The following is an example list of files and directories that are needed when installing an LPP using the **installp** command. The LPP being installed in the following example is **MERGE**.

```
./lpp_name
./usr/lpp/MERGE/liblpp.a

./usr/lpp/MERGE/lpp.loc
./usr/lpp/MERGE/lpp.loc/al.loc
./usr/lpp/MERGE/lpp.loc/inst_updt.loc
./usr/lpp/MERGE/lpp.loc/lpp.hist
./usr/lpp/MERGE/lpp.loc/info

./usr/lpp/MERGE/lpp.loc/files
./usr/lpp/MERGE/lpp.loc/files/local
./usr/lpp/MERGE/lpp.loc/files/local/MERGE1
./usr/lpp/MERGE/lpp.loc/files/local/MERGE2
./usr/lpp/MERGE/lpp.loc/files/local/MERGE3
./usr/bin/MERGE
```

13.6.6 Important Example Files

FILE: **./lpp_name**

```
   Merge                DOS Merge
```

FILE: **./usr/lpp/MERGE/liblpp.a**

This archive file contains the following files:

al

copyright

instal

lpp.hist

FILE: The **al** file in **./usr/lpp/MERGE/liblpp.a**

```
   ./usr/lpp/MERGE/lpp.loc/inst_updt.loc
   ./usr/lpp/MERGE/lpp.loc/al.loc
   ./usr/lpp/MERGE/lpp.loc/lpp.hist
   ./usr/lpp/MERGE/lpp.loc/info
   ./usr/lpp/MERGE/lpp.loc/files/local/MERGE1
   ./usr/lpp/MERGE/lpp.loc/files/local/MERGE2
   ./usr/lpp/MERGE/lpp.loc/files/local/MERGE3
   ./usr/bin/MERGE
```

This file is the apply list for this installation. In order to install files in the local file system, the directory **./usr/lpp/MERGE/lpp.loc** must exist and contain those files.

FILE: The **copyright** file in **./usr/lpp/MERGE/liblpp.a**

```
   This is LPP MERGE
   MERGE LPP
   Version 00.00.0000.0000 (C) Copyright Vendor
```

FILE: The **instal** file in **./usr/lpp/MERGE/liblpp.a**

```
   /etc/inurest -q -d$1 /usr/lpp/MERGE/al MERGE
   exit $?
```

This is an executable file invoked by **installp**. It is written by the vendor.

FILE: The **lpp.hist** file in **./usr/lpp/MERGE/liblpp.a**

```
   c MERGE     00.00.0000.0000 ......  ........ MERGE
   t MERGE LPP
```

This is the history file for the replicated root file system.

FILE: **./usr/lpp/MERGE/lpp.loc/lpp.hist**

```
   c MERGE   00.00.0000.0000 ......  ........ MERGE
   t MERGE   LPP for local
```

This is the history file for the local.

FILE: **./usr/lpp/MERGE/lpp.loc/al.loc**

```
./local/MERGE1
./local/MERGE2
./local/MERGE3
```

This is an apply list file for the local file system.

FILE: **./usr/lpp/MERGE/lpp.loc/inst_updt.loc**

```
cdir=`pwd`
cd /
/etc/inurest -q -d$cdir/bac $cdir/al.loc MERGE
exit $?
```

This is an executable for the local installation. All cluster sites have the LPP installed. If a site-specific installation were required, this script would need to be changed by the vendor.

FILE: **./usr/lpp/MERGE/lpp.loc/info**

```
a MERGE 60 00.00.0000.0000
```

This information file is required for the local installation. For a description of this file, refer to "The Local Information File" in topic 13.10.

13.7 What You Need to Update an LPP

This section describes the type of information, the files and the directories that you need to update an LPP.

Subtopics

13.7.1 The Four Steps In the update Procedure

There are four steps involved in the **update** procedure:

apply

commit

uncommit

reject

apply allows for a conditional or temporary installation. **commit** allows for a more permanent installation. **uncommit** brings the files back to the **apply** conditional status. **reject** returns the changed LPP to its condition before the changes were made.

Figure 13-8 shows the relationship between the four steps in the **update** procedure.

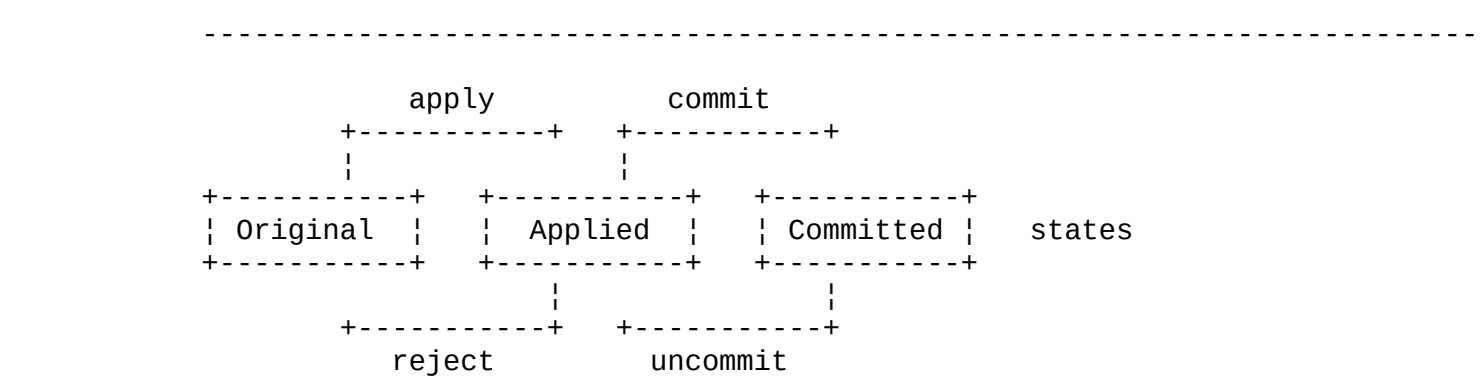


Figure 13-8. Four Steps in the update Procedure

Each of the states is described in the following section.

13.7.2 Original

This is the state before a particular **update** is done. It can occur after an initial installation. It can also occur after a **commit** or **reject** has been done on a previous **update**.

13.7.3 Original to Applied

The Applied state exists after an **apply**. **apply** recovers files from the media, possibly overlaying old files. To save the old files in **/usr/lpp/***lpp-id***/inst_updt.save**, **inusave** may be called from **instal**.

In the Applied state, the directories **/usr/lpp/***lpp-id* and **/usr/sys/inst_updt** contain the LPP files and status that are required for a **commit**.

13.7.4 Applied to Committed

The Committed state exists after a **commit**, which moves the **/usr/lpp/***lpp-id***/inst_updt.save** and **/usr/sys/inst_updt** directories to another location (the backout stack). It also allows for a new **apply** to be performed. A **commit** is performed if a particular LPP needs to be stored in a condition that is more permanent than an **apply**.

Note: Another **update** (be it a subsequent **update** or an **update** to a different LPP) cannot be applied until any outstanding **update** has been committed (or rejected). An **update** in the **applied** state is a temporary state until the system administrator **commits** or **rejects** it.

13.7.5 Committed to Applied

This transition reverses a **commit** by retrieving the **/usr/lpp/***lpp-id* and **/usr/sys/inst_updt.save** directories from a frame stack in storage. It then restores the directories to their conditions prior to the **commit**.

13.7.6 Applied to Original

This transition reverses an **apply** by using the information kept in the **/usr/lpp/***lpp-id* and **/usr/sys/inst_updt** directories to reconstruct the old version of the LPP. When this occurs, the directory **/usr/sys/inst_updt** and some files in **/usr/lpp/***lpp-id* used for **reject** are deleted.

Note: All changes done by **updatep** are logged in the history file and the global history file.

Subtopics
13.7.6.1 Files for Updates

13.7.6.1 Files for Updates

To update an LPP on the AIX system using the **updatep** command, supply the following files in backup-by-name format, using a relative path name with respect to the **/** directory on the target system. The files must be in the following order:

1. An optional file named **./copyright** that contains the copyright information for each LPP on the media. If this file is present, it must be the first. It is referenced by the external media copyright label if there is not enough space on the label to include all of the appropriate copyright notices.

2. A file named **./lpp_name** that contains the title of each LPP (see "The LPP Name File" in topic 13.9.6).

3. An archive named **./usr/sys/inst_updt/control**, created with the **ar** command, that contains the following files:

   ***lpp-id*_vrl**
   A file for each LPP to be updated that contains the version, release and level numbers of the LPP after the update has been applied. The format of the one-record file is:

   ```
   VV RR LLLL
   ```

   This format is similar to all VRL numbers used in this section, except that there are spaces instead of periods between the version, release, and level. The record ends with a new-line character.

   lppsize
   A required file that contains an entry for each LPP that is being updated. Each entry contains:

   lpp-id size

   The **size** field specifies the number of 1024-byte blocks needed to save the changed portion of the LPP specified by the **lpp-id** field. The two fields must be separated by a blank. Each entry ends with a single new-line character.

   The **updatep** service tool uses the size information to determine if there is enough space in the **/usr** file system to save changed and deleted files. If the size of the LPPs being added is larger than the available free space in the **/usr** file system, **updatep** gives the user the following options:

   Stop the update

   Continue with the update.

   If the user continues, **updatep** does a **commit** rather than an **apply** of the update. The save step is skipped; thus, the current version of the LPP cannot be recovered. A subsequent **uncommit** and **reject** of the update will not work.

   Although this file is optional, you should include an **lppsize** file in all LPP updates. If **updatep** does not find

this information, it does not check to see if there is enough space to save the previous version of the LPP.

***lpp-id*_instr**
An optional archive, one for each LPP being updated, that contains a set of files with instructions. If you do not provide instruction files, do not include this archive. These are standard text files. They are named according to the update level to which they apply, in the following format:

```
ui.VV.RR.LLLL
```

In this format, the symbols **VV**, **RR** and **LLLL** represent the version, release, and level numbers. You can have instruction files for one or more of the LPP levels that are being updated.

***lpp-id*_erata**
An optional archive, one for each LPP being updated, that contains a set of files with changes to the documentation for the LPP. If you do not provide changes to the documentation, do not include this archive. These files are standard text files. They are named according to the update level to which they apply, they are in the following format:

```
me.VV.RR.LLLL
```

In this format, the symbols **VV**, **RR** and **LLLL** represent the version, release and level numbers. You can have documentation change files for one or more of the LPP levels being updated.

4. One file named **./usr/sys/inst_updt/special** that identifies special update requirements for each LPP being updated. See "The Special File" in topic 13.10.1 for information about this file.

5. An archive named **./usr/lpp/***lpp-id***/inst_updt/arp**, created with the **ar** command, that contains the following files:

update
An executable file that contains a procedure to update the LPP. The procedure may be either a shell script or a compiled program.

al_*VV.RR.LLLL*
An apply list file that contains the relative path names with respect to the **/** directory of all files to be updated. See "Creating an Apply List File" in topic 13.9.7 for information about the file format. Include one apply list file for each update to the LPP. The symbols **V** , **R** and **L** represent version, release and level numbers as described in "Creating the LPP History File" in topic 13.9.1.

lpp.acf
An optional file that defines archiving procedures for the LPP as described in "The Archive Control File" in topic 13.9.8.

copyright
An optional file that contains appropriate copyright information for the LPP. If this file is present, **updatep** displays the contents of this file when it applies the update to the LPP.

appscr.*nnnn* and rejscr.*nnnn*

where **nnnn** is a **level** number. These are optional executable files. They are used to apply and reject any parts of the update that cannot be accomplished by simple file creation and replacement. If either file exists, they must both exist.

6. The new and replacement LPP files.

13.7.7 Files For Updates to Local File Systems

The following files and directories are required for upgrade updates that include changes to local files.

./usr/lpp/*lpp-id***/lpp.loc**
A directory containing the following files:

inst_updt.loc Executable which does the local installation.

info A file containing installation information and local file system requirements.

lpp.hist A history file for the local installation.

./usr/lpp/*lpp-id***/lpp.loc/files**
A directory containing the local updated files.

In addition, the **arp** archive must contain the file **loc_*VV.RR.LLLL***. This is the local apply list.

Local Examples: Assume you have an LPP named "PCI" and you want to update two local files, **<LOCAL>bin/pci1** and **<LOCAL>sys/pci2**. In order to do this, you must have the following entries in the file **loc_*VV.RR.LLLL*** in the **arp** archive:

./bin/pci1

./sys/pci2

You must also have the following files on your media:

./usr/lpp/PCI/lpp.loc/files/local/bin/pci1

./usr/lpp/PCI/lpp.loc/files/local/sys/pci2

13.7.8 Example update Procedure

Figure 13-9 is a shell procedure to update an example LPP. The parameters have the following meaning:

$1 The full path name of the apply list file.

$2 The device path, usually **/dev/rfd0**.

The name of the LPP must be substituted for **lpp-id**.

```
  NAME=lpp-id
  LDIR=/usr/lpp/$NAME/inst_updt
  AL=$1
  RESTDEV=$2

  #
  #       Save the files to be updated
  #

  /etc/inusave $AL $NAME
  RC=$?
  if [$RC -ne 0 ]; then
      exit $RC
  fi

  #
  #       Restore the updates
  #
  /etc/inurest "-d$RESTDEV" $AL $NAME
  RC=$?

  #
  #       Exit
  #
  exit    $RC
```

Figure 13-9. Example of update Shell Script

13.7.9 Example of Update Files and Directories

The following is a list of files and directories residing on the **updatep** media for an update. The LPP to be updated in this example is called **MERGE**. There are several files to be updated on the local filesystem, as well as on the replicated root. Following the update, MERGE will be at version 0, release 0, level 20.

./lpp_name
./usr/sys/inst_updt
./usr/sys/inst_updt/control
./usr/sys/inst_updt/special

./usr/lpp/MERGE
./usr/lpp/MERGE/inst_updt
./usr/lpp/MERGE/inst_updt/arp

./usr/lpp/MERGE/lpp.loc
./usr/lpp/MERGE/lpp.loc/info
./usr/lpp/MERGE/lpp.loc/lpp.hist
./usr/lpp/MERGE/lpp.loc/files
./usr/lpp/MERGE/lpp.loc/files/local
./usr/lpp/MERGE/lpp.loc/files/local/merge_L1
./usr/lpp/MERGE/lpp.loc/files/local/merge_L3
./usr/lpp/MERGE/lpp.loc/files/local/merge_L4
./usr/lpp/MERGE/lpp.loc/files/local/merge_L5
./usr/lpp/MERGE/lpp.loc/files/local/merge_L6
./usr/bin/merge_g1
./usr/bin/merge_g3
./usr/bin/merge_g4
./usr/bin/merge_g5
./usr/bin/MERGE
./usr/bin/MERGE/mergeX

FILE: **./lpp-name**

MERGE Merge LPP

FILE: **./usr/sys/inst_updt/control**

This is an archive file containing the **lppsize** and **MERGE_vrl** files.

ARCHIVE MEMBER: **lppsize**

MERGE 20

ARCHIVE MEMBER: **MERGE_vrl**

00 00 0020 0000

FILE: **/usr/syst/inst_updt/special**
This file is empty

FILE: **./usr/lpp/MERGE/inst_updt/arp**

This is an archive file containing the files **copyright**, **update**, **al_*00.00.0020***, and **loc_*00.00.0020***.

If apply and reject scripts were needed, archive members **appscr.*0020***

and **rejscr.*0020*** would also be present. Local apply and reject scripts, if needed, would be in **lappscr.*0020*** and **lrejscr.*0020***.

ARCHIVE MEMBER: **copyright**

```
    (c) Copyright AIX PS/2 DOS Merge 1988
    (c) Copyright International Business Machines Corporation 1987
    All Rights Reserved
```

ARCHIVE MEMBER: **update**

This is an executable file that is invoked by **updatep** to do the actual update.

```
    NAME=MERGE
    LDIR=/usr/lpp/$NAME/inst_updt

    AL=$1
    RESTDEV=$2

    #
    #       Save the files to be updated
    #

    /etc/inusave $AL $NAME
    RC=$?
    if [$RC -ne 0 ]; then
      exit $RC
    fi

    #
    #       Restore the updates
    #
    /etc/inurest "-d$RESTDEV" $AL $NAME
    RC=$?

    #
    #       Exit
    #
    exit    $RC
```

ARCHIVE MEMBER: **21_00.00.0020** (apply list for root)

```
  ./usr/bin/merge_g1
  ./usr/bin/merge_g2
  ./usr/bin/merge_g3
  ./usr/bin/merge_g4
  ./usr/bin/merge_g5
  ./user/bin/MERGE/mergeX
```

FILE: **loc_00.00.0020** (apply list for local)

```
  ./local/merge_L1
  ./local/merge_L3
  ./local/merge_L4
  ./local/merge_L5
```

FILE: **./usr/lpp/MERGE/lpp.loc/info**

```
    a MERGE 60

FILE: ./usr/lpp/MERGE/lpp.loc/lpp.hist

This is the <LOCAL> history file.

    a MERGE          00.00.0020.0000 ...... ........ MERGE  Localpr  updat

FILE: ./usr/lpp/MERGE/lpp.loc/inst_updt.loc

This is the script that performs the local upgrade.

    cdir=`pwd`
    cd /
    /etc/lpp/insvl "$cdir"/al.loc MERGE
    /etc/inurest -q -d$cdir/bac $cdir/al.loc MERGE
    exit $?
```

13.8 Allowing for Recovery

The **updatep** service tool requires that the **update** script save enough information to allow it to recover the previous level of the LPP being updated. For simple updates calling **inusave** before trying to change anything is sufficient. The **inusave** program saves the current version of any file.

The **update** service tool may either restore or delete the saved files, depending upon how the update turns out. It deletes the saved files if:

The user commits (accepts) the update (the files are actually move elsewhere).

An error occurs during the update process, but before any files hav been changed by the update process.

It recovers the saved files if:

The user rejects the updat

An error occurs during the update process after files have bee changed.

If an error occurs after the update process has already changed any file, the saved files are not automatically deleted after recovery. The **/usr/lpp/lpp-id/inst_updt.save** directory and its contents must be removed locally. Otherwise, a later install or update for that LPP will detect the presence of that directory, causing the installation to fail.

A file **/etc/lpp/config.log** containing status information will be created at the beginning of each **installp** and **updatep**, and will be removed before the service tool exits. If the machine goes down in the middle of an operation, this file will not be removed.

Error recovery is performed using **/etc/lpp/inuconfig**. This program must be invoked if an error occurs during use of the service tools. It checks for the existence of **/etc/lpp/config.log**. If the file does not exist, this means that the previous service was successfully accomplished. If the file does exist, **inuconfig** will notify the user about a possible bad system configuration caused by the previous service and fix any problem it can find.

13.9 File Formats and Description

This section provides detailed file format information for:

The LPP History fil

The Global History fil

The Local History fil

The LPP Requirements fil

The Requirement Strin

The LPP Name fil

The Apply List fil

The Archive Control file

Subtopics

13.9.1 Creating the LPP History File

The LPP history files **/usr/lpp/***lpp-id***/lpp.hist** and **/usr/lpp/***lpp-id***/lpp.loc/lpp.hist** contain information to identify the installed release and version of an LPP on the system. These are ASCII text files. See the ***AIX Operating System Technical Reference*** for further information.

Figure 13-10 shows the record format of the history files. Figure 13-11 defines the fields in the history record.

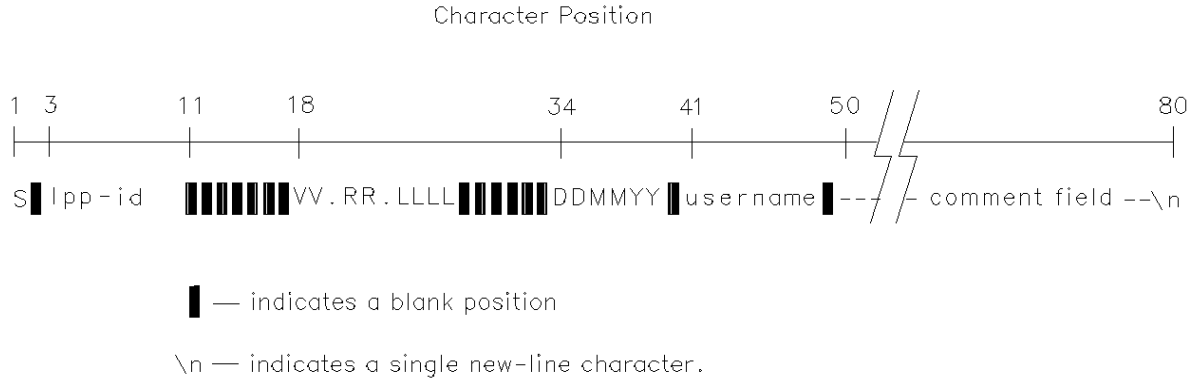


Figure 13-10. Record Format for the History File

Figure 13-11. Fields in a History Record

Field	Description
S	Indicates the condition of the LPP, using the following characters: **a** An update was applied. **b** Positions 3-16 contain the name of the backup format file used to install this LPP. **c** An update was committed (accepted). **e** An unrecoverable error has occurred. **m** A manual reject was performed. The state of the LPP is unknown. **r** An update was rejected or recovered. **t** This record is a title record. Positions 3 through 32 contain a title for the LPP. ***** This record is a comment field (put an * in position 79 to ensure a full length record).
lpp-id	The name assigned to the LPP. If the name is less than 8 characters, the field must be filled out with blanks.
VV	A 2-digit numeric field followed by a period indicating the version number of the LPP. The version number indicates which level of the hardware and operating system the LPP works with.
RR	A 2-digit numeric field followed by a period indicating the

release number of the LPP. The release number tracks changes to external programming interfaces since the last version change. This number increments each time the external interface to the LPP changes.

LLLL A 4-digit numeric field indicating the update state of the LPP. This field increments when the LPP changes and the change does not affect the documented external interface for the LPP. The level, together with the **S** field, ensures that all changes up to and including the current change are installed on the system.

DDMMYY These three numeric fields indicate the date the change is made to the LPP:

DD	Day of the month (01 to 31)
MM	Month of the year (01 to 12)
YY	Year (00 to 99)

username An alphanumeric field that contains the user ID of the person that installed the LPP. If the user ID is less than 8 characters, this field must be filled out with blanks. This field is filled in at installation time, and can be blank when the update is distributed.

comment field A 30-character field for adding comments.

\n A required new-line character.

13.9.2 The Global History File

The global history file, **/etc/lpp/ghf** contains a chronological log of all installations and updates for all LPPs on the system. This file provides a snapshot of current status of software on the system. This file is unique to each system.

The format of this file is similar to **lpp.hist** except that the status field is 2 characters long. This field contains an L if the local component exists, or a blank space if it does not exist. All other fields are shifted to the right by 1. Each line in this file must be eighty characters long. All access to this file is done through **inudhist**. This file is modified as a result of the following:

When a successful **installp** is done, a log of the successful installation is made through **inudhist -i** (a **commit** entry).

When an **updatep -a (apply)** is done, a log of the **apply** is made through **inudhist -a (apply entry)**.

When an **updatep -c (commit)** is done, the previous apply record is commented out with a "*" in the first column and a new commit record is added through **inudhist -c**.

When an **updatep -u (uncommit)** is done, the previous **commit** record is commented out with a "*" in the first column and a new apply record is added through **inudhist -u**.

When an **updatep -r (reject)** is done, the last record (being an **apply**) is commented out with a "*" in the first column using **inudhist -r**.

13.9.3 The Local History File

The purpose of the local history file is to log relevant information on changes made to the local filesystem of an individual site. Each site has a local history file which is kept under **<LOCAL>/lpp.hist**. These files are not used for determining what local changes are pending for a site as that is taken care of by the queue mechanism. Therefore, the local history file is only of interest to the system administrator as a log of the site's past, but not as a means of controlling any LPP.

The format for this history file is identical to the global history file.

Kernel information can be recorded in this history file as a comment. The comment line syntax for the history file must adhere to the following rules:

An asterisk (*) in column 1

To ensure an 80 byte record, place an asterisk in column 79 before th line feed.

This history file will be modified as a result of a local action (pending in the queue) being processed by **qproc**.

13.9.4 Creating the LPP Requirements File

You can use the **ckprereq** routine to determine if required LPPs are already installed on the system. The **ckprereq** routine uses the LPP requirements file (**prereq** by default) to make that determination. Figure 13-12 shows the record format of an entry in the requirements file. Figure 13-13 defines the fields in the requirements record.

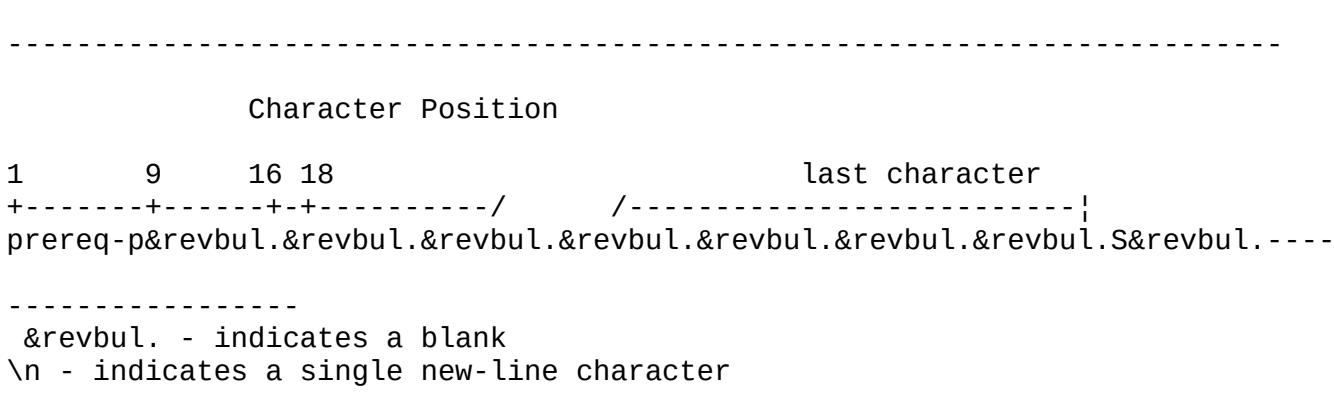


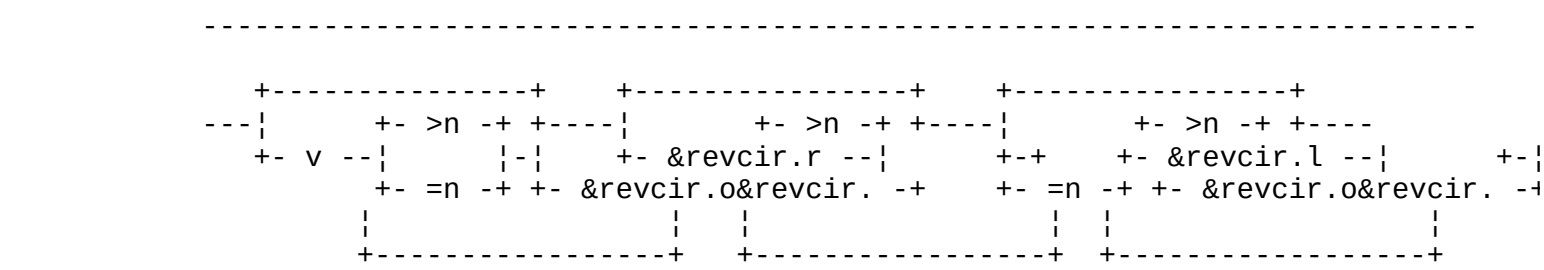
Figure 13-12. Record Format for the Requirements File

Figure 13-13. Fields in Requirements Record

Field	Description
prereq-p	An 8-character alphabetic field that contains the name of the LPP that is required.
S	**ckprereq** result code: Leave this field blank. The **ckprereq** command fills in this 1-character field with one of the following letters: **f** The format of the history file is not 80-byte records. **l** The requested level is not installed on the system. **n** A history file for that LPP is not installed on the system. **r** The requested release is not installed on the system. **s** The **prereq** file entry has a syntax error. **u** The state of the LPP is unknown. **v** The requested version is not installed on the system. **blank** The prerequisite LPP is installed on the system in the proper configuration.
requirement string	A set of logical expressions that define the version, release and level parameters that the prerequisite LPP must have. "Using the Requirement String" in topic 13.9.5 explains how to use the requirement string.
\n	Required new-line character.

13.9.5 Using the Requirement String

The requirement string allows you to specify a minimum version, release and level required for an LPP to be serviced. Figure 13-14 shows the syntax of the requirement string and the symbols used to create the string.



&revcir. - indicates required blank
n - can be any integer

Figure 13-14. Format for the Requirement String

The symbols in the diagram have the following meanings:

v A version number is being specified.

r A release number is being specified.

l A level number is being specified.

> Greater than.

= Equal.

o Logical OR.

n An integer value for the specific parameter to be compared against.

The **ckprereq** program evaluates each requirement string in the **prereq** file beginning with the **v** (for version) and continuing to the right (left to right order). It does not evaluate any further than is required to determine that the whole line must be true or false. For example, if the LPP **myprog** is at version 3.1 (version 3, release 1), and the requirement string is:

```
  myprog           v>1 r>0
```

The program needs to determine only that version 3 is larger than 1 to be sure that the correct conditions exist. If, however, **myprog** is at version 2.1, the routine must not only evaluate the version but also the release. Version 2.0 would fail the test; version 2.1 passes the test.

Subtopics
13.9.5.1 Requirements File Example Entry

13.9.5.1 Requirements File Example Entry

The following entry:

```
  myprog           v=2 r>30 o =10 o =15
```

in a requirements file indicates that:

> The LPP **myprog** is a required LPP.
> It must be version 2
> Any level is valid *l* is not specified).
> The release may be either 10, 15 or greater than 30

Similarly, the following entry:

```
  myprog           v>1 r>30 o =10 o =15
```

in a requirements file indicates that:

> The LPP **myprog** is a required LPP.
> It must be version 2 or greater
> Any level is valid *l* is not specified).
> If the version is 2, the release may be either 10, 15 or greater tha
> 30; if the version is 3 or more, any release is valid.

13.9.6 The LPP Name File

The LPP name file contains the names of the LPPs on the service media. Name this file **./lpp_name**. It must contain one or more entries.

Figure 13-15 shows the format of an entry in the LPP name file and Figure 13-16 defines the fields in the entry.

```
-------------------------------------------------------------------------

          Character Position

                                                 47 or
1       9       17                               last character
+-------+-------+-----------/     /----------¦
lpp-id  &revbul.&revbul.&revbul.&revbul.&revbul.&revbul.&revbul.&revbul.-- tit

-----------------
 &revbul. - indicates a blank
\n - indicates a single new-line character

-------------------------------------------------------------------------
```

Figure 13-15. Format for the LPP Name file

Figure 13-16. Fields in LPP Name Entry

Field	Description
lpp-id	The name assigned to the LPP. If the name is less than 8 characters, this field must be filled out with blanks.
title	A descriptive title of up to 30 characters.
\n	A required new-line character.

13.9.7 Creating an Apply List File

Apply List

Apply lists are files containing a number of file names, one per line, using relative path names that begin with **./** (for example, **./usr/bin/tail**). Apply lists will typically contain names of files that are functioning ports of the LPP. Sometimes they must also contain service tool control files, as is described below.

These lists are intended to be used by the restore program to bring files on to the system. Due to the manner in which restore operates, the only directories in the apply lists should be those specific to the LPP (no **./etc** or **./usr/lib**). If a directory is present on the media and is in the apply list, its attributes (fstore, owner, etc.) on the system will be changed. This is usually not desired. A directory that is not on the system nor on the media will be created with default attributes if needed.

Note: The apply list for an install and the merged set of apply lists for an update must not be larger than 14K-bytes.

Global apply list

These apply lists are always used while the current working directory is **/**, so all file names should be relative to **/**.

For **installp**, the global apply list is a member of the **/usr/lpp/lpp-id/liblpp.a** archive called **al**. If there is any local installation to be performed, this list must include all files in the **/usr/lpp/lpp-id/lpp.loc** directory tree.

For **updatep**, the global apply list is a set of members of the **/usr/lpp/lpp-id/inst_updt/arp** archive, named **al_vrl** for upgrade updates (one for each separate level). Any new archive member files that are provided for in the **lpp.acf** file must appear in the apply list(s) as well.

Local apply list

These apply lists are always used while the current working directory is **/local**, so all file names should be relative to **/local**.

For **installp**, the local apply list is **/usr/lpp/lpp-id/al.loc**. This list would typically include all files found in the **/usr/lpp/lpp-id/lpp.loc/files/local** directory tree.

For **updatep**, the local apply list is a set of members of the **/usr/lpp/lpp-id/inst_updt/arp** archive, named **loc_vrl** for upgrade updates (one for each separate level). These lists would typically include relevant files from the **/usr/lpp/lpp-id/lpp.loc/files/local** directory tree.

13.9.8 The Archive Control File

The service tools use the optional archive control file to place individual member files into libraries.

Supply a file named **lpp_acf** in the **liblpp.a** library for an install or in the **arp** library for an update. This archive control file lists the new library member files and the libraries they belong in. Each line has the following format:

filename archive_file

The parameters on each line are separated by one or more blanks, and have the following meaning:

filename The relative full path name of the new library member exactly as it appears in the apply list.

archive_file The full rooted path name of the library file that contains the member file.

13.10 The Local Information File

The **/usr/lpp/***lpp-id***/lpp.loc/info** file contains information relevant to the local service of the LPP. It is required for service. For example, for the *lpp-id* **MERGE**, the file may contain:

```
  a MERGE 60 00.00.0020.0000
```

Note the following:

Each field is separated by a space and there is no specified fiel size

The first field describes the action (this must be "a" for **apply**)

The second field is the LPP name **MERGE**)

The third field is the number of 1K-byte blocks required to servic the LPP on the local.

The fourth field describes the version, release, and level

Subtopics
13.10.1 The Special File

13.10.1 The Special File

The Special file must be present and must be empty. It is reserved for future use.

13.11 The Save and Recover Directory

This section describes the directory and files for the root replicated file system.

The save and recover directory (**/usr/lpp/***lpp-id***/inst_updt.save**) contains copied files and extracted archived files that were saved during a reinstallation, or during application of an update. If a reinstallation procedure created the directory, the installation procedure must run **inurecv** if an error occurs, or delete the directory when installation is complete. If an update procedure created the directory, the directory exists until the update is either committed or rejected, or until **updatep** recovers the directory because of an error during the update procedure. At that time **updatep** deletes this directory, after recovering the saved files if the update was rejected. However, if the update is rejected without automatic recovery, the directory is not deleted to allow for manual recovery of the saved files. The directory must then be manually deleted after the files are recovered.

For local files, the directory is similar, but is is found under **/local/lpp/***lpp-id***/inst_updt.save** instead of **/usr/lpp/***lpp-id***/inst_updt.save**. All other features are the same.

The directory contains the files listed in Figure 13-17. The order is not important. The directory can contain only the file names in the list.

Figure 13-17. Save/Restore Directory Content

File	**Contents**
lpp-id	A file in **backup/restore** format containing copies of all files affected by this update. Pathnames are in root-relative format (e.g., **./usr/bin/MERGE**).
archive.list	An optional file that lists any archived files that were extracted and saved in this directory as described in **archive.n** below. This file consists of one record for each archive file saved. This record has the following format: archive.**n** **member-name archive-name**\n Where **member-name** is the full path name of the file when it is restored on the system, **archive-name** is the full path name of the target archive file where this file belongs, and **\n** represents a single new-line character.
archive.*n*	A file named in this form for each record in the **archive.list** file. The file contains the file that was saved as listed in the **archive.list** file. In this form for naming the file: **archive** Identifies the file as a backed up archive file. *n* Is an integer. Each backed up file has a unique **n** associated with it, starting with **1** for the first file backed up. For example, three backed up files have the names **archive.1**, **archive.2** and **archive.3**.

uniq_dir.list An optional file that lists any install or update directories that were created during the process of archiving library members during an update. It permits **updatep** to delete any **inst_updt/libname** directories when committing or rejecting an update (except for the directories **/usr/sys/inst_updt** and **/usr/lpp/***lpp-id***/inst_updt** which are used by the update process). The file consists of one record for each member file in the directory that is created. The record consists of the full path name of the member file:

/**path**/inst_updt/**libname/member**

In this form the parts have the following meanings:

path	The full path name to the **inst_updt** directory
libname	The name of the library that was created
member	The name of the member file

13.12 Internal Commands

The following commands are used as part of the service process. They are available for use in the various procedures created as part of this process, (.e.g., **update**, **lappscr.vrl**, etc.).

Figure 13-18. Internal Commands

Command	Description
/etc/ckprereq	Checks the revision level of LPPs.
/etc/errupdate	Adds, replaces, or deletes the error report format templates in the file **/etc/errfmt**.
/etc/inudocm	Gets copies of update instructions or book changes to look at or print.
/etc/inurecv	Recovers all files and archived member files that a previous execution of **inusave** saved. This command also recovers any other saved files recorded in the configuration list file, or active list file.
/etc/inurest	Does simple restores and archives.
/etc/inusave	Saves some, or all, of the files and archived member files that will be replaced or modified when the LPP is installed or updated.
/etc/inuupdt	Applies a maintenance update for a single LPP.
/etc/trcupdate	Adds, replaces, or deletes trace report format templates in the file **/etc/trcfmt**.
/etc/lpp/insvl	The local equivalent to inusave. Works on target directory in **/local/lpp**.
/etc/lpp/inrcvl	The local equivalent to inurecv. Works on target directory in **/local/lpp**.
/etc/lpp/qapp	Appends a new local action to the system queue.
/etc/lpp/qproc	Triggers the processing of all pending queue entries on an individual site.
/etc/lpp/ckstack	Checks the backout stack to find the frame containing the desired backout. If there are other frames (containing other lpps which must be backed out before the desired lpp due to prerequisites), it will create a file containing a list of other frames. The file created is called **/usr/lpp.save/list**.

14.0 Chapter 14. Maintaining Different Versions of a Program

Subtopics

14.1 CONTENTS

14.2 About This Chapter

This chapter shows how to use the Source Code Control System (SCCS) to control revisions to source code or documentation files using the major SCCS commands. It provides background information about SCCS, the format of SCCS files, and how to use the SCCS commands. In addition, it shows a sample SCCS session and describes the three major SCCS commands in detail. These commands are **admin**, **get**, and **delta**. The other commands are described in *AIX Operating System Commands Reference*.

14.3 Introducing SCCS

The ***Source Code Control System*** (SCCS) allows one person or a group to control and account for changes made to source code or documentation files. It stores the changes made to a file instead of storing the changed file. This allows several versions of the same file to exist in the system. To edit the file, specify the version. SCCS builds that version based on its stored information about previous changes made. Using SCCS reduces storage requirements and helps track the development of a project that requires keeping many versions of large programs.

Subtopics

14.3.1 Features

The SCCS commands form a complete system. Once you create an SCCS file, use an SCCS command to change it. Do not edit or compile the SCCS file itself. Use another file that is derived from the original SCCS file for these operations.

SCCS commands can do the following:

Create an SCCS fil

Get a version of an SCCS fil

Save changes made to that file versio

Define who can change an SCCS fil

Record who made changes to the SCCS fil

Record when and why the changes were made

14.3.2 New Terms

The following descriptions of SCCS terms are used in this chapter.

SCCS file Any file containing text (source code or documentation) that is controlled with SCCS commands. All SCCS files begin with **s.**. This file contains the original file contents and sets of changes to the original file or later versions of that file. It also contains information about who can change the file, who made changes and when they were made.

Note: Do not edit this file directly. It contains information to build the stored files.

delta A set of changes made to an SCCS file. After changing a file, use the **delta** command to save those changes in the SCCS file, thereby creating a new delta. Create a new delta only to save the changes made. When editing a specific version of an SCCS file, that version may consist of several different deltas.

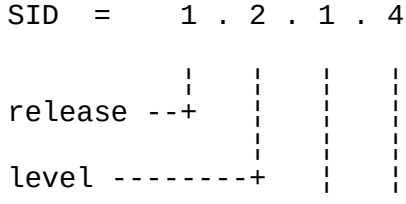
SID ***SCCS Identification***: The name assigned to a delta. An SID has up to four parts as shown in Figure 14-1.

Every SCCS file starts out with an SID of 1.1. which means release 1, level 1. After editing version 1.1 and saving the changes, SCCS gives the new delta an SID of 1.2, which means release 1, level 2.

A typical SCCS file only uses release and level numbers and grows in a straight line. In these cases, the latest file version uses every previous delta to that file. However, a file may branch to a path where a file version consists of a subset of all of the deltas. For example, a common file can be used by two different groups. Both groups need the same code up to a certain point, and then each group goes its own way. In this case, create a branch delta that allows each group to add deltas onto a common base.

The file then has a trunk, with deltas identified by release and level, and one or more branches, which have deltas containing all four parts of an SID. On a branch, the release and level numbers are fixed and new deltas are identified by changing sequence numbers. Note that a file version built from a branch does not use any deltas placed on the trunk after the point of separation. See Figure 14-2.

SCCS can combine different deltas. However, when combining many deltas into one, SCCS loses track of the changes that created each of the smaller (old) deltas and only tracks the change to create the larger (new) delta.



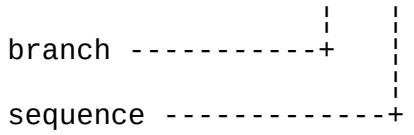


Figure 14-1. Parts of an SID

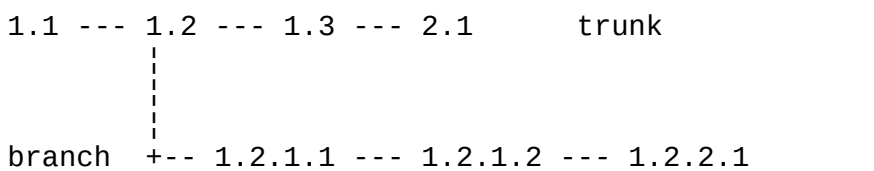


Figure 14-2. Growth of an SCCS File with Branching

14.3.3 SCCS File Format

SCCS files have two major sections called the header and the body. The header has five subsections, which identify who created the file, who can change it and other administrative details. The body has one or more subsections consisting of the text portions of the file. There is one text portion for each delta in the SCCS file.

The **get** command uses the header and body to create the specified file version. Other SCCS commands make use of the header and body to perform their functions.

Never edit the SCCS file itself except to modify incorrect information in the header as described in "Locating Damaged SCCS Files" in topic 14.4.1.1. Editing the SCCS file can damage the structure of the file. The SCCS commands can do any necessary modifications of the header or body. To see how an SCCS file is organized, examine an SCCS file using simple commands such as **cat** or an editor. However, do not edit the SCCS file.

Subtopics
14.3.3.1 The SCCS File Header
14.3.3.2 The SCCS File Body

14.3.3.1 The SCCS File Header

The sections of an SCCS file header are described in the following.

Checksum A number containing the logical sum of all of the characters in the file. The **admin** command uses the checksum to ensure that all changes to the file were made by using the SCCS system.

Delta Table Information about each delta including type, SID, date and time of creation, and comments.

User Names List of login names or group IDs of users who are allowed to modify the SCCS file by adding or removing deltas. If this parameter does not exist, anyone can modify the SCCS file.

Options List of indicators that control specific actions of various SCCS commands.

Comments Descriptive text provided by the user to describe the contents or purpose of the file.

14.3.3.2 The SCCS File Body

The SCCS File body includes the actual text of the deltas in the file. The body also contains SCCS control text intermixed with the delta text. Notice that the deltas are in reverse order. That is, the most recently created delta is the first one in the list.

Warning: Using non-SCCS commands with SCCS files can damage the SCCS files. Changing an SCCS file with a non-SCCS command makes the checksum incorrect. See "Locating Damaged SCCS Files" in topic 14.4.1.1 for information about working with the checksum.

14.3.4 Command Conventions

In most cases, SCCS commands accept the following two types of parameters:

flags Flags begin with the **-** (minus sign), followed by a lowercase character, and sometimes followed by a value. They control how the command operates.

arguments Arguments may be file or directory names. They specify the file or files with which the command operates. Using a directory name as an argument specifies all SCCS files in the directory.

Arguments cannot begin with a **-** (minus sign). If you specify the **-** (minus sign) by itself, the command reads standard input until it reaches end of file character (usually signalled by the user pressing **Ctrl-D**). This can be useful when using pipes. When using the keyboard for input, it reads until end of file.

Any flags specified for a command apply to all files specified for that command and are processed before arguments to that command. Their placement in the command line is not important. Arguments are processed left to right. Some SCCS files contain flags that determine how the command operates on the file. See "Using the admin Command" in topic 14.4.1 for more information.

SCCS Commands produce error messages with the following format:

```
ERROR [file]: message text (code)
```

The code in parentheses can be used as an argument to the **sccshelp** command. The **sccshelp** command can sometimes provide more information about a particular error code.

An SCCS command stops processing a file that contains a fatal error. Any other files in the command are still processed.

14.3.5 Command Summary

The following summary presents the commands in the order of their use. They are further defined in *AIX Operating System Commands Reference*.

admin Creates an SCCS file or changes some characteristic of an existing SCCS file.

get Gets a specified version of an SCCS file. Use this command to get a copy of a file to edit or compile.

unget Undoes the effect of a previous use of the **get -e** command.

delta Adds a set of changes (delta) to the text of an SCCS file.

rmdel Removes a delta from an SCCS file. The delta must be the most recent delta on its branch.

cdc Changes the comments associated with a delta.

what Searches a system file for a pattern and displays what follows it. Use this command to find identifying information.

sccsdiff Shows the differences between any two versions of an SCCS file.

comb Combines two or more consecutive deltas of an SCCS file into a single delta. Combining deltas may reduce storage requirements.

val Checks an SCCS file to see if its computed checksum matches the figure listed in the header.

prs Prints portions of an SCCS file in a specified format.

sccshelp Provides an explanation of a diagnostic message.

vc Substitutes assigned values in place of keywords.

sact Displays current SCCS file editing status.

14.4 Using SCCS Commands

Figure 14-3 shows how to create, change and update the contents of an SCCS file. The SCCS commands have much more function than what is shown in the figure. System responses to the SCCS commands are not shown.

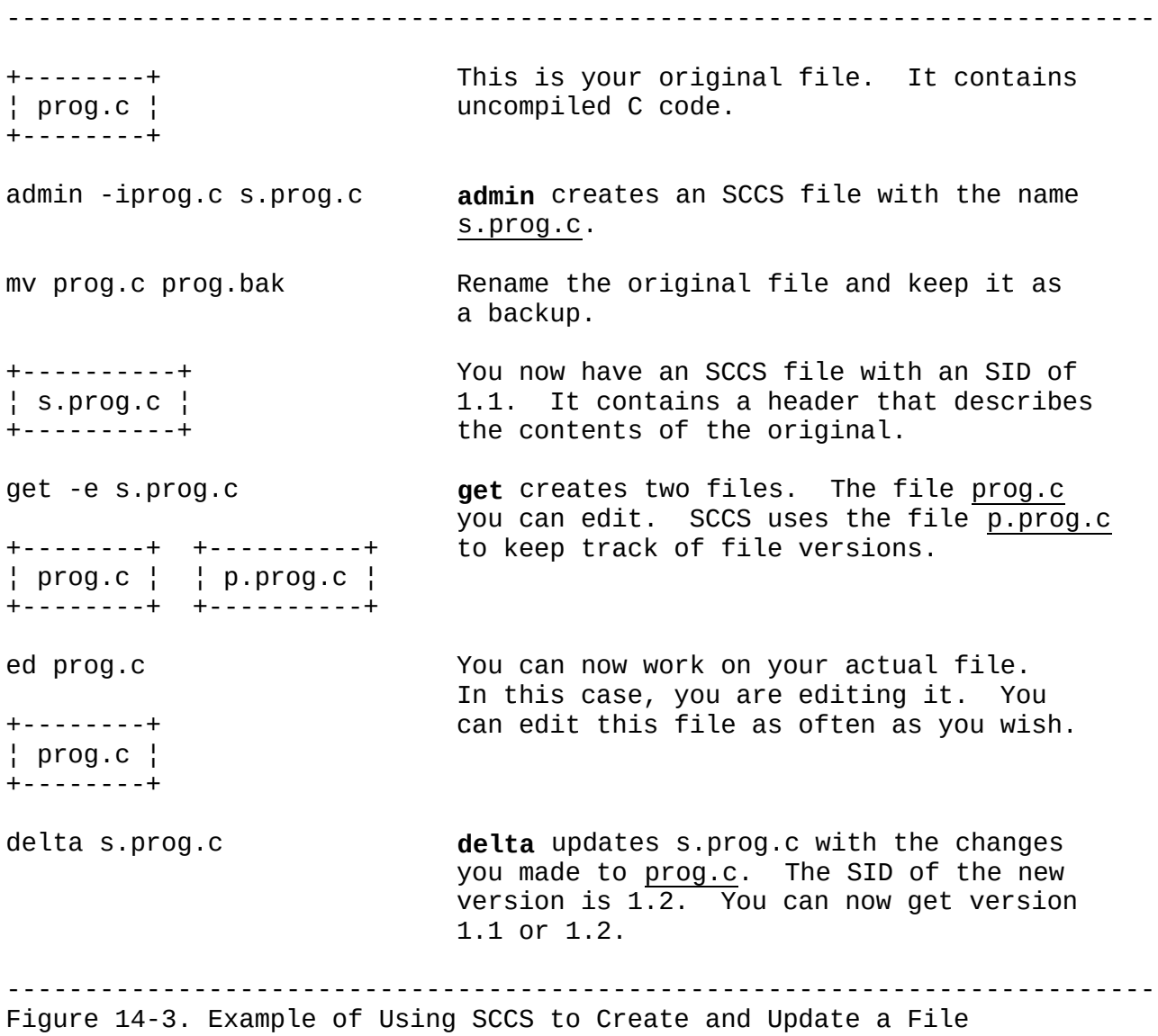


Figure 14-3. Example of Using SCCS to Create and Update a File

Subtopics
14.4.1 Using the admin Command
14.4.2 Using the get Command
14.4.3 Using the delta Command
14.4.4 Using the sccshelp Command

14.4.1 Using the admin Command

These examples use an imaginary text file called **test.c**, and an editor such as **ed** to edit files.

First, create an ordinary SCCS file. If you use the **-i** flag, **admin** creates delta 1.1 from the specified file. Without the **-i flag**, **admin** creates an empty SCCS file. Once delta 1.1 is created, rename the original text file so it does not interfere with SCCS commands. For example, to create an SCCS file from a file **test.c**:

```
admin -itest.c s.test.c
No id keywords (cm7)
li
s.test.c test.c
```

Then rename the original text file:

```
mv test.c  back.c
```

The message, **No id keywords (cm7)** does not indicate an error. SCCS writes this message when there are no identification keywords in the file. Identification keywords are variables that can be placed in an SCCS file. The values of these variables provide information, such as date, time, SID, or file name. See "Getting Read-Only File Versions" in topic 14.4.2.1 for an explanation of identification keywords. If there are no identification keywords, SCCS writes the message.

Name the SCCS file anything as long as it begins with **s.**. In the above example, the original file and the SCCS file have the same name, but that is not necessary.

Because you did not specify a release number, **admin** gave the SCCS file an SID of 1.1. SCCS does not use the number 0 to identify deltas. Therefore, a file cannot have an SID of 1.0 or 2.1.1.0. All new releases start with level 1. To start the **test.c** file with a release number of 3, use the **-r** flag with the **admin** command, as shown below:

```
admin -i test.c -r3 s.test.c
```

To restrict permission to change SCCS files to a specific set of user IDs, list their user IDs or group ID numbers in the user list of the SCCS file by using the **-a** flag of the **admin** command. These IDs then appear in the SCCS file header. Without the **-a** flag to restrict access, all user IDs can change the SCCS files.

```
admin -adan s.test.c
```

Subtopics
14.4.1.1 Locating Damaged SCCS Files

14.4.1.1 Locating Damaged SCCS Files

Although SCCS provides some error protection, you may need to recover a file that was accidentally damaged. This damage may result from a system malfunction, operator error, or changing an SCCS file without using SCCS commands.

SCCS commands use the checksum to determine whether a file was changed since it was last used. The only SCCS command that processes a damaged file is the **admin** command when used with the **-h** or **-z** flags. The **-h** flag tells **admin** to compare the checksum stored in the SCCS file header against the computed checksum. The **-z** flag tells **admin** to recompute the checksum and store it in the file header.

Check SCCS files on a regular basis for possible damage. The easiest way to do this is to run the **admin** command with the **-h** flag on all SCCS files or SCCS directories as shown below:

```
admin -h s.file1  s.file2 ...

admin -h directory1  directory2  ...
```

If **admin** finds a file where the computed checksum is not equal to the checksum listed in the SCCS file header, it displays this message:

```
corrupted file (co6)
```

If a file was damaged, try to edit the file again, or read a backup copy. After fixing the file, run the **admin** command with the **-z** flag and the repaired file name:

```
admin -z s.file1
```

This operation replaces the old checksum in the SCCS file header with a new checksum based on the repaired file contents. Other SCCS commands can now process the file.

14.4.2 Using the get Command

The **get** command gets files in either ***read-only*** or editable form. You can only use the editable form to create a delta to the SCCS file. Use the read-only form to print or compile the file. The results of using **get** on a file depend on whether you specify the file as read-only or editable. The examples are divided into read-only examples and editable examples. The examples are not related unless the comments specify that they are related.

Note: You must use the **-e** flag with the **get** command, to create a delta.

Subtopics
14.4.2.1 Getting Read-Only File Versions
14.4.2.2 Getting Editable File Versions
14.4.2.3 Getting Duplicate File Versions

14.4.2.1 Getting Read-Only File Versions

To compile a program or print a document from an SCCS file, get the file as read-only. The **get** command performs different tasks when it gets a read-only document.

The difference between the two types of **get** operations is important when using identification keywords in a file. Identification keywords can appear anywhere in a file. They are symbols that are replaced with some text value when **get** retrieves the file as read-only. For example, to print the current date and SID in a file, put the following symbols in the file:

```
%H%  %I%
```

%H% is the symbol for the current date and **%I%** is the symbol for the SID. When **get** retrieves a file as editable, it leaves the symbols in the file and does not do text value substitution. See *AIX Operating System Commands Reference* for the identification keywords to use in a file.

Several examples of the **get** command are shown below:

To check the file directory, enter:

```
li              /* check file directory */
```

The system responds with:

```
s.test.c
```

To get file test.c, enter:

```
get s.test.c    /* get file test.c */
```

The system responds with:

```
3.5
59 lines
```

To check the file directory, enter:

```
li              /* check file directory */
```

The system responds with:

```
s.test.c test.c
```

Because you did not specify a version of the file, **get** built the version with the highest SID. In the next two examples, the **-r** flag specifies which version to get:

To get a specific version of file test.c, enter:

```
get -r1.3 s.test.c
```

The system responds with:

```
1.3
40 lines
```

To get a specific version of file test.c, enter:

```
get -r1.3.1.4 s.test.c
```

The system responds with:

```
1.3.1.4
50 lines
```

If you specify just the release number of the SID, **get** finds the file with the highest level within that release number.

To get the highest level of file test.c within a release number, enter:

```
get -r2 s.test.c
```

The system responds with:

```
2.7
21 lines
```

If the SID specified is greater than the highest existing SID, **get** gets the highest existing SID. If the SID specified is lower than the lowest existing SID, SCCS writes an error message. In the following example, release 7 is the highest existing release:

To get the highest existing SID of file test.c, enter:

```
get -r9 s.test.c
```

The system responds with:

```
7.6
400 lines
```

The **-t** flag gets the top version in a given release or level. The top version is the most recently created delta, independent of its location. In the next example, the highest existing delta in release 3 is 3.5, while the most recently created delta is 3.2.1.5.

To get the highest existing delta , enter:

```
get -t -r3 s.test.c
```

The system responds with:

```
3.2.1.5
46 lines
```

14.4.2.2 Getting Editable File Versions

All of the previous examples use the **get** command to get a read-only file. To edit the file and create a new delta, use the **-e** flag. The **get** command works differently when using the **-e** flag, so the previous examples may not apply. Restrictions for files built with the **-e** flag are explained in the *AIX Operating System Commands Reference* under the **get** and **admin** commands. If you build the wrong version of the file, use **unget** to undo the effect of the **get -e** command.

Several examples of the **get** command are shown below.

To check the file directory, enter:

```
li                /* check file directory */
```

The system responds with:

```
s.test.c
```

To get an editable version of file test.c, enter:

```
get -e s.test.c   /* get editable version of test.c */
```

The system responds with:

```
1.3
new delta 1.4
67 lines
```

To check the file directory, enter:

```
li                /* check file directory */
```

The system responds with:

```
p.test.c  s.test.c  test.c
```

The working file is **test.c.**. If you edit **test.c** and save the changes with the **delta** command, SCCS creates a new delta with an SID of 1.4. The file **p.test.c** is a temporary file used by SCCS to keep track of file versions.

In the previous example, you could use the **-r** flag to get a specific version. Assuming delta 1.3 already exists, the following three uses of the **get** command are the same:

```
get -e s.test.c
get -e -r1 s.test.c
get -e -r1.3 s.test.c
```

To start using a new (higher in value) release number, get the file with the **-r** flag and specify a release number greater than the highest existing release number. In the next example, release 2 does not yet exist.

To get a new release of file test.c, enter:

```
get -e -r2 s.test.c
```

The system responds with:

```
  1.3
  new delta 2.1
  67 lines
```

Notice that **get** indicates the version of the new delta that will be created if the **delta** command stores changes to the SCCS file. If the example did not include the **-e** flag, **get** would build the highest existing SID (1.3) and would not indicate a new delta, even though the **-r2** flag requests a version **2.1**.

To develop a version of the SCCS file that does not depend on the most recently created delta, create a branch delta as shown in Figure 14-4.

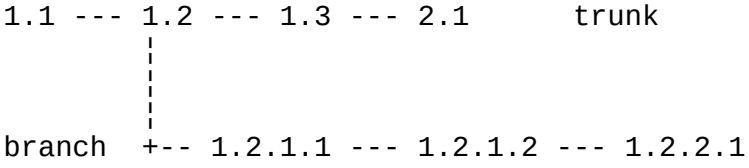


Figure 14-4. Growth of an SCCS File with Branching

In the figure, a branch exists at version 1.2. Adding new deltas at two different places in the SCCS file creates programs for two environments that have some similar code (1.1 and 1.2) and some code that is not the same.

Creating another branch from delta 1.2 builds a second series of program files. The new branch begins with delta **1.2.2.1**.

To create a branch delta, use the **-r** flag and specify the release and level where the branch occurs. In the next example, deltas 1.3 and 1.4 already exist.

To create a branch delta, enter:

```
  get -e -r1.3 s.test.c
```

The system responds with:

```
  1.3
  new delta 1.3.1.1
  67 lines
```

Create deltas on branches using the same methods.

14.4.2.3 Getting Duplicate File Versions

To edit a file, get the file version using the **get** command (with the **-e** flag) and save the changes with the **delta** command. Several different editable versions of an SCCS file can exist as long as each one is in a different directory. If you try to get the same editable file version more than once without using the **delta** command, SCCS writes an error message.

To get the same editable file version more than once, set the **j** option in the SCCS file with the **admin** command. Set the **j** option using the **-f** flag. You can then get the same SID several times from different directories, creating a separate file for each **get** command. Although the files originate from a single SID, SCCS gives each of them a unique new SID.

To obtain the starting directory, enter:

```
pwd
   /u/dan/sccs       /* starting directory   */
```

To set the j option, enter:

```
admin -fj s.test.c    /* set the j option     */
```

To get the latest version of file test.c, enter:

```
get -e s.test.c       /* get latest version   */
```

The system responds with:

```
1.1
new delta 1.2
5 lines
```

To change to directory new, enter:

```
cd /u/new          /* change to directory new  */
```

To get version 1.1 again, enter:

```
get -e /u/dan/sccs/s.test.c  /* get 1.1 again  */
```

The system responds with:

```
1.1
new delta 1.1.1.1
5 lines
```

Notice that SCCS creates two deltas, 1.2 and 1.1.1.1, from the single original file version of 1.1. Look at the **p.test.c** file. It shows a separate entry for each version currently in use. The **p.test.c** file remains in the directory until you take care of both file versions with either the **delta** command or the **unget** command.

14.4.3 Using the delta Command

The **delta** command saves the changes made to a particular version of an SCCS file. To use the **delta** command, follow these instructions:

1. Use **get -e** to get an editable version of the file.

2. Edit that file.

3. Use **delta** to create a new version of the SCCS file.

When using the **delta** command, it prompts for comments. The comments are for that particular delta and appear in the SCCS file header. The comments are not retrieved when you **get** the delta and do not appear in the text of a retrieved file. Use comments to keep track of why a delta was created.

To see the comments, use an editor to look at the SCCS file, write the SCCS file to the display screen with the **cat** command, or print selected parts of the file to standard output using the **prs** command. Refer to *AIX Operating System Commands Reference* for descriptions of these commands. Remember not to change the contents of the SCCS file directly. To change the delta comments, use the **cdc** command.

A common use of the **delta** command is shown below:

```
delta s.test.c
Enter comments, terminated with EOF or blank line:
```

Then enter comments, as follows:

```
This delta contains the payroll function
```

delta then finishes processing and displays:

```
1.4
24 inserted
3 deleted
45 unchanged
```

The above example stores the comment in the SCCS file header and creates delta 1.4. It then lists how many lines of text are inserted, deleted, or unchanged. SCCS may give unexpected numbers for these categories because of its definition for an edited line of text. However, the number of lines inserted plus the number of lines left unchanged should equal the total number of lines in the file. SCCS does not allow using the **delta** command if an editable file does not exist. However, once an editable file exists (created with **get -e**), SCCS creates the delta without checking the data being stored in the file.

Note: When using identification keywords in SCCS files, do not use the **delta** command with a file built as read-only if an editable version of the file also exists. When you get a file as read-only, SCCS replaces identification keywords with their values. Using the **delta** command on the file saves the values and the identification keywords are lost. To recover, remove the delta, or re-edit the file and replace the identification keywords.

14.4.4 Using the sccshelp Command

SCCS provides a limited form of help for certain error codes and all of the SCCS commands. To get help on a specific command or error code, use the following format:

```
sccshelp [command].. [code]..
```

The **sccshelp** program prompts for a **command** or an **error code** if those parameters are not included in the command. If it does not have information about a specific error code, **sccshelp** writes an error message and continues processing. For example, to get help on **rmdel** and two error codes, enter the following:

```
sccshelp rmdel gee ge5
```

The **sccshelp** command replies:

```
rmdel:
        rmdel -r<SID> <file> ...

ERROR: gee not found (he1)

ge5:
"nonexistent sid"
The specified sid does not exist in the given file.  Check for typos.
```

The response indicates that either the **sccshelp** command does not have information for the error code **gee** or the code does not exist.

15.0 Chapter 15. Finding and Changing Strings

Subtopics

15.1 CONTENTS

15.2 About This Chapter

This chapter contains introductory information about some of the system programs that are useful when developing programs. These programs are not required to create programs, but they do provide added services that make checking and maintaining programs easier. The commands described in this chapter:

Find a specified series of characters in a text fil
Find and change information in a text fil
Make fast editing changes on a large text file

Complete reference information about all commands is in *AIX Operating System Commands Reference*.

The examples in this chapter contain ASCII characters; however, the commands also recognize National Character Languages. For additional information on Multibyte Character Set (MBCS), see the discussion in Chapter 18, "International Character Support" in topic 18.0.

15.3 Finding Strings

The system provides three similar programs to help locate a series of characters (***string***) in a file. These programs are:

grep A general function program that finds literal strings in a file, and also finds strings that you specify using wildcard characters. (These are not the same as the shell wildcard characters).

fgrep A faster version of **grep** that only finds literal strings in a file (wildcards are not allowed).

egrep An extended version of **grep** that looks for more complex expressions.

Use these programs to search one or more files at a time to answer the following questions:

In which file(s) does the string occur **-l** flag)?

On how many lines in each file does the string occur **-c** flag)?

What is the line number of each place that the string occurs **-n** flag)?

What lines in which file(s) contain the string (no flag)

What lines in which file(s) exactly match the string **-x** flag)?

What lines in which file(s) do not contain the string **-v** flag)?

What is the 512-byte disk block of the file in which the string occur (**-b** flag)?

Subtopics
15.3.1 Strings
15.3.2 Example of Commands

15.3.1 Strings

A ***string*** is any group of characters to find. Enclose the string in ' ' (single quotes) to ensure that the shell does not interpret blanks or special shell characters in the string as part of its syntax.

For example, because blanks separate the parameters on the command line, the shell interprets the command:

```
fgrep find me myfile
```

as a request to find the string, **find**, in the files **me** and **myfile**. To find the string, **find me**, in **myfile** specify the command like:

```
fgrep 'find me' myfile
```

Subtopics
15.3.1.1 Literal Strings
15.3.1.2 Regular Expressions

15.3.1.1 Literal Strings

A literal string is a string that does not contain wildcard characters, and can, therefore, be interpreted just as it is. The previous example contains a literal string, **find me**. Use literal strings to specify exactly what to find.

For example, to find the file that contains the module, **eprog()**, in five source files that contain several modules each, use the command:

```
fgrep 'eprog()' file1 file2 file3 file4 file5
```

This command finds not only the actual module but any references to it. Look at the output lines to determine which reference actually contains the module definition. To find only the module definition, use the **-x** flag:

```
fgrep -x 'eprog()' file1 file2 file3 file4 file5
```

For coding formats that put each module definition left-justified on a line by itself, this command writes out only that line, together with the name of the file in which it occurs.

15.3.1.2 Regular Expressions

A regular expression is a string that contains wildcard characters and operators that define a set of one or more possible strings. Programs that find strings use a set of wildcard characters that is different from the shell wildcards, but the same as the line editor, **ed**. These wildcard characters and operators are:

.(period)
Specifies any character except new-line.

^ (caret)
Specifies the beginning of the line when it is the first character in a regular expression.

$ (dollar sign)
Specifies the end of the line when it is the last character in a regular expression.

[ ] (square brackets)
Encloses a set of characters (not empty) that represents any **one** of the characters in the set.

[abc] Represents either a or b or c.
[a-c] Is the same as [abc] (**grep** only). The hyphen defines a range of ASCII values that starts with the value of the first letter and ends with the value of the second letter.
[A-z] Defines more than just the letters of the alphabet (**grep** only). It defines the range of ASCII values from A (065) to z (122).

|
OR - indicates a search for either one string or another:

```
egrep 'prog()|prog1()' file1
```

finds lines containing either **prog()** or **prog1()**.

In addition to the regular expression support described above, AIX also supports searches for regular expressions containing international characters. Refer to Chapter 18 of this book for details on the "Use of Regular Expressions with International Characters." Refer to the **ed** command in *AIX Operating System Commands Reference* for details on building regular expressions.

15.3.2 Example of Commands

To check a C program for proper nesting of braces, use the following two commands:

```
grep -c '{' file1
```

and

```
grep -c '}' file1
```

Each command responds with a number that represents the number of lines in the file containing either { or }, respectively. If the numbers are not the same, you may have a problem. Check the file again using a different form of the command:

```
egrep '{|}' file1
```

This command displays each line in the file that contains either a { or a }. It displays the lines in the order that they occur in the file, so that you can quickly check for matching pairs of open and closed braces.

15.4 Scanning Files

The **awk** program is an extension of the features of **grep**. It performs the following operations:

Scans a file or list of files to find matches to a ***regular expression***

Performs an operation on the lines that are found, defined by a ***action***.

It uses all of the regular expression building features that **egrep** uses, plus it allows you to:

Write selected fields of the lin

Calculate running totals

The **awk** program is a procedural language that finds and changes strings in text files. In addition, it provides numeric processing, variables, more general pattern selection for finding strings, and flow control statements. This program treats both string and numeric data. In general, this program is useful for:

Processing input to find numeric counts, sums or subtotal

Verifying that the contents of a field contains only numeri information

Checking to see that delimiters are balanced in a programming fil

Processing data contained in fields within line

Changing data from one program into a form that can be used by different program

Changing syntax in a program source fil

Changing system calls when porting from one system to another

Subtopics

15.4.1 Program File

When using **awk**, you can either enter the search pattern directly on the command line as with **grep**, or you can build a file that contains both the search pattern and the actions to perform. Using a program file puts many patterns in one file, and saves typing the command again to correct an error in the search pattern. When using a program file, run **awk** with the **-f** flag, such as:

```
awk -f pfile file1 file2 file3
```

In this command, **pfile** is the name of the program file, **file1**, **file2**, and **file3** are the files to be searched, and **-f** tells **awk** to look in **pfile** for the search program.

The program file is a series of statements that look like:

```
pattern         { action }
pattern         { action }
pattern         { action }
   .
   .
   .
```

Where:

pattern Is a regular expression, or series of regular expressions, that defines the search pattern, including:

- Boolean combinations of regular expressions using the operators **!** **||** **&&** and **()** .
- Boolean combinations of relational operators on strings, numbers, fields, variables, and array elements.

action Is a set of steps to perform on the line, designated with **awk** commands and operators, including:

- Any expressions that are used in a ***pattern***
- Arithmetic and string expressions
- **Assignment** statements
- **If-else** statements
- **While-for** statements
- More than one output stream.

{ } Are delimiters that set off the action from the search pattern.

In any line, you can omit either the pattern or the action. If you omit the pattern, **awk** performs the action on all lines in the file(s); if you omit the action, **awk** copies the line to standard output.

When **awk** runs, it reads the first line of the input data file and matches it against each of the ***patterns*** in the program file in the order that they appear in the program file. When **awk** finds a pattern that matches the line, it performs the associated ***action*** on that line. Then it continues to search for more matches in the program file. When it has compared the first input line against all patterns in the program file, **awk** reads the next input line and starts at the beginning of the program file with that line.

15.4.2 Variables

Awk recognizes the following built-in variables:

FILENAME	The name of the current input file.
NR	The number of the current record.
NF	The number of fields in the current record.
FS	The character used for a field separator.
RS	The character used for a record separator.
$0	The contents of the input record.
$n	The contents of field **n** of the input record.
OFS	The character used for output field separator (the character between fields when the data is written; a blank if you do not change it).
ORS	The character used for output record separator (the character between records when the data is written; a new-line if you do not change it).

15.4.3 BEGIN and END

The **awk** program recognizes two special keywords that define the beginning (**BEGIN**) and the end (**END**) of the input file. The pattern **BEGIN** matches the beginning of the input before reading the first record. Therefore, **awk** performs any actions associated with this pattern once, before processing any of the input file. **BEGIN** must be the first pattern in the program file. For example, to change the field separator to a colon for all records in the file, include the following line as the first line of the program file:

```
BEGIN   {FS=":"}
```

Similarly, the pattern, **END**, matches the end of the input file after processing the last record. Therefore, **awk** performs any actions associated with this pattern once, after processing all of the input file. **END** must be the last pattern in the program file. For example, to print the total number of lines in the input file, include the following line as the last line in the program file:

```
END     {print NR}
```

15.4.4 Using Regular Expressions as Patterns

The simplest regular expression is a literal string of characters, enclosed in slashes. For example, if the program file contains only the entry:

```
/the/
```

the file is a complete program that displays all lines containing the string **the**. Because the string does not specify any blanks or other qualifiers, the program also displays lines containing words such as:

```
theater
northern
```

that have the string as part of them. The program is sensitive to uppercase and lowercase, and only displays lines containing the lowercase form of the string.

Subtopics
15.4.4.1 Character Class
15.4.4.2 Special Characters

15.4.4.1 Character Class

To find lines that contain **The** (the string as the first word in a sentence) in addition to the lowercase version, use a character class to represent either uppercase or lowercase. A ***character class*** is a set of characters, enclosed in **[ ]** (square brackets). Each character in the character class satisfies the search conditions for **one** character position. For example, to find lines containing both forms of the word **the** (and words containing it), use the string:

```
/[Tt]he/
```

The following string:

```
/[cCdDhH]ome/
```

finds lines that contain the words:

```
come
Come
dome
Dome
home
Home
```

Use ranges within a character class to indicate a group of consecutive ASCII characters. To define a range, enter the following:

1. **[**
2. The first character of the range
3. **-** (minus)
4. The last character of the range
5. **]**

Note: Ranges specify a continuous sequence of ASCII character codes, not alphabetic order. For example, the range [Z-a] specifies only eight characters from ASCII code 90 (Z) to 97 (a).

15.4.4.2 Special Characters

The **awk** program defines the symbols shown in Figure 15-1 to use in building patterns:

Figure 15-1. awk Special Characters

Symbol	Meaning
/	String delimiter: Indicates the start and end of a string or regular expression.
\	Escape: Tells **awk** to treat the next character as a regular ASCII character instead of a symbol that **awk** treats as a special character.
$0	Matches the entire line with the pattern.
$n	Matches field **n** (**n** is an integer) in each input line.
~	Match field operator: Tells **awk** to match the regular expression with a specified field in each line, not the line.
!~	Not match field operator: Tells **awk** to compare the regular expression with a specified field in each line and perform the action only if the expression does not match the field.
^	Beginning of the line or field: When placed at the start of a string, tells **awk** to match the string only when it occurs at the start of a line or specified field.
$	End of the line or field: When placed at the end of a string, tells **awk** to match the string only when it occurs at the end of a line or specified field.

15.4.5 Using Relational Expressions as Patterns

Use a relational expression as a pattern in the program file. The **awk** program defines the following relational operators for use in building patterns:

<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
==	Equivalent
!=	Not equivalent

Subtopics
15.4.5.1 Examples of Relational Expressions in a Pattern

15.4.5.1 Examples of Relational Expressions in a Pattern

To find lines that contain an even number of fields:

```
NF % 2 == 0
```

To find lines that begin with **s, t, u,...**:

```
$1 >= "s"
```

To perform a string comparison between the first two fields:

```
$1 > $2
```

15.4.6 Using Combinations of Patterns

Combine two or more patterns using Boolean operators:

oror	Or
&&	And
!	Not

For example, the pattern:

```
  $1 >= "T" && $1 < "U" && $1 != "The"
```

finds lines that begin with T, and are not the word **The**.

15.4.7 Using Pattern Ranges

A pattern range allows the use of one pattern to begin an action on the lines of text, and another pattern to end the action on lines of text. Specify a pattern range by using two patterns separated by commas. The first pattern specifies the starting pattern; the second pattern specifies the ending pattern. Therefore, the line:

```
/The/,/End/  {action}
```

finds the first line that contains the pattern **The** and begins performing the **action** on all lines following it in the file until **awk** finds a line containing the pattern **End**. **awk** does not change either the line containing **The** or the line containing **End**.

Similarly, the line:

```
NR==100,NR==200       {action}
```

performs the action starting at line 100 and ending at line 200 of the input file.

15.4.8 Using Functions in an Action

The **awk** program provides the following functions to use within an action:

length Returns the length of the current record.

length(arg) Returns the length of the string specified by **arg**.

sqrt(arg) Returns the square root of **arg**.

log(arg) Returns the base e logarithm (natural logarithm) of **arg**.

exp(arg) Returns the exponential part of **arg**.

int(arg) Returns the integer part of **arg**.

substr(s,m,n) Returns a string that is part of string **s**, beginning at character **m** and continuing for **n** characters (or the end of string **s**). If **m** is 1, the string starts at the beginning of string **s**. If you do not supply a value for **n**, the string continues to the end of string **s**.

index(s1,s2) Returns the character position in string **s1** where string **s2** occurs. If **s2** is not in **s1**, this function returns a zero.

sprintf(f,e1,e2,...) Returns a formatted string. The parameters are:

f A formatting specification string defined using the formatting specifications of the **printf** library routine.

e1,e2,... A series of strings that the **f** format specification acts upon.

The function formats the argument strings (**e1, e2, ...**) using the format specification **f**, and returns the formatted string.

15.4.9 Using Variables in an Action

The **awk** program sets all variables in actions to zero when it begins executing the action. The variables do not have a strict type; they take on numeric (floating point) values or string values depending on their use in the action expression. For example, the expression:

```
x = 1
```

indicates that **x** is a numeric variable. Similarly, the expression:

```
x = "smith"
```

indicates that **x** is a string variable. However, **awk** converts between strings and numbers when needed. Therefore, the expression:

```
x = "3" + "4"
```

assigns a value of 7 (numeric) to **x**, even though the arguments are literal strings. If **awk** cannot change a string variable to numeric when you are using it as a numeric variable, **awk** assigns it a numeric value of zero. To force a variable to be treated as a single type:

string Add the null string (" ") to the value assigned to the variable.

numeric Add zero (0) to the value assigned to the variable.

15.4.10 Using Operators in an Action

Use the following operators to build expressions within the action statement:

+	Addition
-	Subtraction
*****	Multiplication
/	Division
%	Modulo (remaindering)
++	Increment
--	Decrement
+=	Increment by value
-=	Decrement by value
***=**	Multiply by value
/=	Divide by value
%=	Modulo by value
~	Match string
!~	Not match string

For example, to find the sum of all the first fields and the sum of all the second fields in a file with the program file:

```
                {s1 +=$1; s2 += $2}
  END           {print s1,s2}
```

15.4.11 Using Field Variables in an Action

Fields in **awk** share the same properties as variables. They can be used in arithmetic or string operations and may be assigned to a numeric or string value. For example, to replace the first field with a sequence number:

```
{$1 = NR;print}
```

To accumulate two fields into a third field:

```
{$1 = $2 + $3;print $0}
```

Use numeric expressions for field references, such as:

```
{print $i,$(i+1),$(i+n)}
```

How you use a field determines whether **awk** treats a field as numeric or string. If it cannot tell how the field is used, **awk** treats fields as strings.

awk splits input lines into fields as needed. You can also split any variable or string into fields. For example:

```
n = split(s,array,sep)
```

splits the string **s** into **array[1]...array[n]** and returns the number of elements. If you provide the **sep** argument, it is the field separator. If you do not provide **sep**, the field separator is the character defined by the variable **FS**.

15.4.12 Concatenating Strings

Concatenate strings by placing their variable names together in an expression. For example, the expression:

```
length($1 $2 $3)
```

Returns the length of the first three fields. The expression:

```
print $1 " is " $2
```

Prints the first two fields separated by **" is "**. You can use variables and numeric expressions when concatenating strings.

15.4.13 Using Arrays

You do not need to declare array elements. The command **awk** sets them to zero when first used. Use any value that is not null, including a string value, for a subscript. An example of the numeric subscript is:

```
x[NR] = $0
```

This expression assigns the current input record to the **NR**th element of the array **x**. For an example of using a string subscript, suppose that the input contains fields with values like **apple** or **orange**. Then the program:

```
/apple/         {x["apple"]++}
/orange/        {x["orange"]++}
END             {print x["apple"], x["orange"]}
```

increments counts for the named array elements and prints them at the end of the input.

15.4.14 Using Control Statements

The **awk** language also provides the following control structures as in the C language:

    If-els
    Whil
    Fo
    Brea
    Continu
    Nex
    Exi
    Braces for statement groupin
    Comments

Subtopics
15.4.14.1 If-Else Statement
15.4.14.2 While Statement
15.4.14.3 For Statement
15.4.14.4 Break Statement
15.4.14.5 Continue Statement
15.4.14.6 Next Statement
15.4.14.7 Exit Statement
15.4.14.8 Comments

15.4.14.1 If-Else Statement

The **if-else** statement is exactly like that of the C language. The condition in parentheses of an **if-else** statement is evaluated; if it is true, the statement following the **if** is performed. The **else** part is optional.

15.4.14.2 While Statement

The **while** statement is exactly like that of the C language. For example, to print all input fields, one on each line, use the following program:

```
i = 1
while(i<=NF)
{
        print $i
        ++i
}
```

15.4.14.3 For Statement

The **for** statement is also like that of the C language. For example, the previous **while** example could also be written:

```
for(i=1;i<=NF;++i)
        print $i
```

15.4.14.4 Break Statement

The **break** statement causes an immediate exit from an enclosing **while** or **for** loop.

15.4.14.5 Continue Statement

The **continue** statement causes the next iteration of an enclosing loop to begin.

15.4.14.6 Next Statement

The **next** statement causes **awk** to skip to the next input record and begin scanning the patterns from the top of the program file.

15.4.14.7 Exit Statement

The **exit** statement causes the program to stop as if the end of the input occurred.

15.4.14.8 Comments

Include comments in the **awk** program file to explain program logic. Comments begin with the # character and end with the end of the line. For example:

```
  print x,y       #this is a comment
```

15.5 Editing Files with sed

The **sed** program is a text editor that has similar functions to those of **ed**, the line editor. Unlike **ed**, however, the **sed** program performs its editing without interacting with the person requesting the editing. This method of operation allows **sed** to:

Edit very large file

Perform complex editing operations many times without requirin extensive retyping and cursor positioning (as interactive editors do)

Perform global changes in one pass through the input

The editor keeps only a few lines of the file being edited in memory at one time, and does not use temporary files. Therefore, the file to be edited can be any size as long as there is room for both the input file and the output file in the file system.

Subtopics

15.5.1 Starting the Editor

To use the editor, create a command file containing the editing commands to perform on the input file. The editing commands perform complex operations and require a small amount of typing in the command file. Each command in the command file must be on a separate line. Once the command file is created, enter the following command on the command line:

```
sed -fcmdfile >output <input
```

In this command the parameters mean:

cmdfile The name of the file containing editing commands.

output The name of the file to contain the edited output.

input The name of the file, or files, to be edited.

The **sed** program then makes the changes and writes the changed information to the output file. The contents of the input file are not changed.

15.5.2 How sed Works

The **sed** program is a stream editor that receives its input from standard input, changes that input as directed by commands in a command file, and writes the resulting stream to standard output. If you do not provide a command file and do not use any flags with the **sed** command, the **sed** program copies standard input to standard output without change. Input to the program comes from two sources:

Input stream A stream of ASCII characters either from one or more files or entered directly from the keyboard. This stream is the data to be edited.

Commands A set of addresses and associated commands to be performed, in the general form of:

[**line1** [,**line2**]] command [argument]

The parameters **line1** and **line2** are called ***addresses***. Addresses can be either patterns to match in the input stream, or line numbers in the input stream as explained in "Selecting Lines for Editing" in topic 15.5.3.

You can also enter editing commands along with the **sed** command by using the **-e** flag.

When **sed** edits, it reads the input stream one line at a time into an area in memory called the ***pattern space*** as shown in Figure 15-3. When a line of data is in the pattern space, **sed** reads the command file and tries to match the addresses in the command file with characters in the pattern space. If it finds an address that matches something in the pattern space, **sed** then performs the command associated with that address on the part of the pattern space that matched the address. The result of that command changes the contents of the pattern space, and thus becomes the input for all following commands.

When **sed** has tried to match all addresses in the command file with the contents of the pattern space, it writes the final contents of the pattern space to standard output. Then it reads a new input line from standard input, and starts the process over at the start of the command file.

Some editing commands change the way the process operates. See the **Control** commands in Figure 15-6 in topic 15.5.5.

Flags used with the **sed** command can also change the operation of the command as shown in Figure 15-2.

Figure 15-2. sed Command Flags

Flag	Function
-n	**sed** does not copy all input lines to standard output. Instead, it copies only those lines that editing commands specifically write to standard output. See the **print lines** and **substitution** commands in Figure 15-6 in topic 15.5.5.
-e	**sed** uses the argument that directly follows this flag as an editing command.
-f	**sed** uses the argument that directly follows this flag as the name of the file containing the editing commands. This file

must contain editing commands with each command on a separate line.

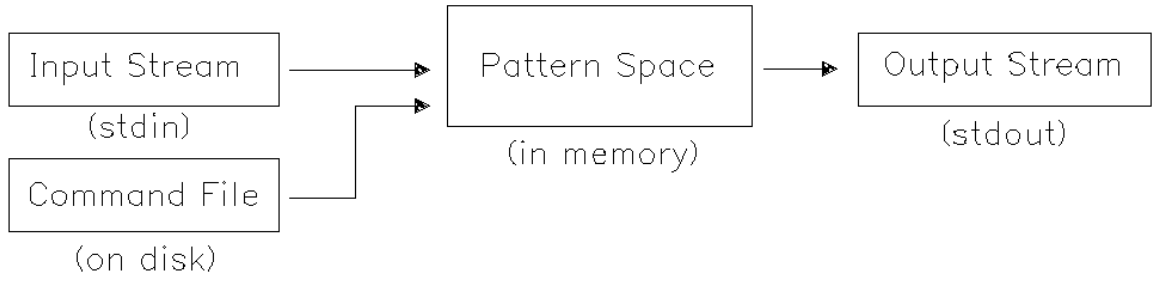


Figure 15-3. sed Block Diagram

15.5.3 Selecting Lines for Editing

Use one of the following forms of addressing to select lines in the input stream for editing:

Line numbers

As the editor reads each input line, it increments its line counter starting with line 1 as the first line of the first file in the input stream. The line counter runs cumulatively through all files in the input stream. The editor does not reset the counter when it opens a new file in the same input stream. The value of the line counter for each line is the ***line number*** for that line.

Specifying a decimal integer as either **line1** or **line2** in the editing commands indicates the line number of the line to be edited. The character **$** matches the last line of the last file in the input stream.

Context addresses

A context address is a ***regular expression*** enclosed in **/** (slashes). See "Regular Expressions" in topic 15.5.4 for a description of the regular expressions that **sed** recognizes. The whole regular expression must match some part of the pattern space for a successful context address match.

Editing commands can have zero, one, or two addresses, depending on the command and how you use it. The number of addresses determines how the address is used:

Addresses	**Use of Command**
No address	The command is applied to every line in the input stream.
One address	The command is applied to each line that matches the address.
Two addresses	The command is applied to the range of addresses starting with the line that matches the first address up to and including the first line that matches the second address. The editor then tries to match the first address again to find another range. Separate two addresses with a comma as shown in the syntax diagrams in Figure 15-6 in topic 15.5.5.

15.5.4 Regular Expressions

A regular expression is a string that contains literal characters, wildcard characters and/or operators that define a set of one or more possible strings. The stream editor uses a set of wildcard characters that is different from the shell wildcards, but the same as the line editor, **ed**. These wildcard characters and operators are shown in Figure 15-4.

Figure 15-4. sed Wildcard Characters

Symbol	**Function**
.	A period specifies any character except new-line.
^	A caret specifies the beginning of the line when it is the first character in a regular expression.
$	A dollar sign specifies the end of the line when it is the last character in a regular expression.
.[ ]	Square brackets enclose a set of characters (not empty) that represents any **one** of the characters in the set. If the first character inside the brackets is a **^** (caret), the regular expression matches any character **except** the characters in the set and the new-line at the end of the pattern space. **[abc]** Represents either a or b or c. **[a-c]** Is the same as [abc]. The hyphen defines a range of ASCII values that starts with the value of the first letter and ends with the value of the second letter. **[A-z]** Defines more than just the letters of the alphabet. It defines the range of ASCII values from A (065) to z (122).
\n	A new-line character matches a new-line character that is not the new-line character at the end of the pattern space.
*****	A regular expression followed by an asterisk matches any number (including 0) of adjacent occurrences of that regular expression.
\(and \)	A set of backslash-parentheses enclose a regular expression that can be repeated using the **\d** expression.
\d	**d** is a single digit. This symbol in **string** is replaced by the set of characters in the input lines that matches the **d**th substring in **pattern**. Substrings begin with the characters **\(** and end with the characters **\)**. See "String Replacement" in topic 15.5.7 for more information about using this expression.
//	The null string is the same as the last regular expression in the edit stream.
****	A backslash tells **sed** to treat the next character as a regular ASCII character instead of a symbol that **sed** treats as a special character.

Refer to the **ed** program in *AIX Operating System Commands Reference* for details for building regular expressions.

15.5.5 sed Command Summary

All **sed** commands are single letters plus some parameters, such as line numbers or text strings. Figure 15-6 summarizes the commands that make changes to the lines in the pattern space. The table uses the symbols shown in Figure 15-5 in the syntax diagrams:

Figure 15-5. Syntax Symbols

Symbol	Meaning
[ ]	Square brackets enclose optional parts of the commands
italics	Parameters in italics represent general names for a name that you enter. For example, **filename** represents a parameter that you replace with the name of an actual file.
line1	This symbol is a line number or regular expression to match that defines the starting point for applying the editing command.
line2	This symbol is a line number or regular expression to match that defines the ending point to stop applying the editing command.

Figure 15-6. sed Command Summary

Category	Function	Syntax/Description
Line Manipulation:	Append lines	**[line1]a\\ntext** Writes the lines contained in **text** to the output stream after **line1**. The **a** command must appear at the end of a line. See "Text in Commands" in topic 15.5.6 for the format of the **text**.
	Change lines	**[line1 [,line2]]c\\ntext** Deletes the lines specified by **line1** and **line2** as the **delete lines** command does. Then it writes **text** to the output stream in place of the deleted lines.
.	Delete lines	**[line1 [,line2]]d** Removes lines from the input stream and does not copy them to the output stream. The lines not copied begin at line number **line1**. The next line copied to the output stream is line number **line2 + 1**. If you specify only one line number, then only that line is not copied. If you do not specify a line number, the next line is not copied. You cannot perform any other functions on lines that are not copied to the output.
	Insert lines	**[line1]i\\ntext** Writes the lines contained in **text** to the output

		stream before **line1**. The **i** command must appear at the end of a line. See "Text in Commands" in topic 15.5.6 for the format of the **text**.
	Next line	**[line1 [,line2]]n** Reads the next line, or group of lines from **line1** to **line2** into the pattern space. The current contents of the pattern space are written to the output if it has not been deleted.
Substitution:	Substitute for Pattern	**[line1 [,line2]]s/pattern/string/flags** Searches the indicated line(s) for a set of characters that matches the regular expression defined in **pattern**. When it finds a match, the command replaces that set of characters with the set of characters specified by **string**. See "String Replacement" in topic 15.5.7 for specifications for this command.
Input and Output:	Print lines	**[line1 [,line2]]p** Writes the indicated lines to **stdout** at the point in the editing process that the **p** command occurs.
.	Write lines	**[line1 [,line2]]w filename** Writes the indicated lines to **filename** at the point in the editing process that the **w** command occurs. If **filename** exists, it is overwritten; otherwise, it is created. A maximum of 10 different files can be mentioned as input or output files in the entire editing process. Include exactly one space between **w** and **filename**.
	Read file	**[line1]r filename** Reads **filename** and appends the contents after the line indicated by **line1**. Include exactly one space between **r** and **filename**. If **filename** cannot be opened, the command reads it as a null file without giving any indication of an error.
Matching Across Lines:	Join next line	**[line1 [,line2]]N** Joins the indicated input lines together, separating them by an imbedded new-line character. Pattern matches can extend across the imbedded new-line(s).

	Delete first line of pattern space	**[line1 [,line2]]D** Deletes all text in the pattern space up to and including the first new-line character. If only one line is in the pattern space, reads another line. Starts the list of editing commands again from the beginning.
	Print first line of pattern space	**[line1 [,line2]]P** Prints all text in the pattern space up to and including the first new-line character to **stdout**.
Pick up and Put down:	Pick up copy	**[line1 [,line2]]h** Copies the contents of the pattern space indicated by **line1** and **line2** if present, to the holding area. The previous contents of the holding area are destroyed.
	Pick up copy, appended	**[line1 [,line2]]H** Copies the contents of the pattern space indicated by **line1** and **line2** if present, to the holding area, and appends it to the end of the previous contents of the holding area.
	Put down copy	**[line1 [,line2]]g** Copies the contents of the holding area to the pattern space indicated by **line1** and **line2** if present. The previous contents of the pattern space are destroyed.
	Put down copy, appended	**[line1 [,line2]]G** Copies the contents of the holding area to the end of the pattern space indicated by **line1** and **line2** if present. The previous contents of the pattern space are not changed. A new-line character separates the previous contents from the appended text.
	Exchange copies	**[line1 [,line2]]x** Exchanges the contents of the holding area with the contents of the pattern space indicated by **line1** and **line2** if present.
Control:	Negation	**[line1 [,line2]]!** The **!** (exclamation point) applies the command that follows it on the same line to the parts of the input file that are **not** selected by **line1** and **line2**.
	Command groups	**[line1 [,line2]]{ grouped commands** } The **{** (left brace) and the **}** (right brace) enclose a set of

	commands to be applied as a set to the input lines selected by **line1** and **line2**. The first command in the set can be on the same line or on the line following the left brace. The right brace must be on a line by itself. You can nest groups within groups.
Labels	**:label** Marks a place in the stream of editing commands to be used as a destination of a branch (see the **b** and **t** commands). The symbol **label** is a string of up to 8 bytes. Each **label** in the editing stream must be different from any other **label**.
Branch to label, unconditional	**[line1 [,line2]]blabel** Branches to the point in the editing stream indicated by **label** (see **:label** above) and continues processing the current input line with the commands following **label**. If **label** is null, branches to the end of the editing stream, which results in reading a new input line and starting the editing stream over. The string **label** must appear as a label in the editing stream.
Test and Branch	**[line1 [,line2]]tlabel** If any successful substitutions were made on the current input line, branches to **label**. If no substitutions were made, does nothing. Clears the flag that indicates a substitution was made. This flag is cleared at the start of each new input line.
Quit	**[line1]q** Stops editing in an orderly fashion by: Writing the current line to the output Writing any appended or read text to the output Stopping the editor.
Find line number	**[line1]=** Writes to standard output the line number of the line that matches **line1**.

15.5.6 Text in Commands

The **append**, **insert** and **change** lines commands all use a supplied text string to add to the output stream. This text string conforms to the following rules:

Can be one or more lines long

Each \n (new-line character) inside **text** must have an additional \ character before it (\\n).

The **text** string ends with a new-line that does not have an additional \ character before it (\n).

Once the command inserts the **text** string, the string:

- Is always written to the output stream, regardless of what other commands do to the line that caused it to be inserted
- Is not scanned for address matches
- Is not affected by other editing commands
- Does not affect the line number counter.

15.5.7 String Replacement

The **s** command performs string replacement in the indicated lines in the input file. If the command finds a set of characters in the input file that satisfies the regular expression **pattern**, it replaces the set of characters with the set of characters specified in **string**.

The **string** parameter is a literal set of characters (digits, letters and symbols). Two special symbols can be used in **string**:

& This symbol in **string** is replaced by the set of characters in the input lines that matched **pattern**. For example, the command:

```
s/boy/&s/
```

tells **sed** to find a pattern **boy** in the input line, and copy that pattern to the output with an appended **s**.

Therefore, it changes the input line:

From: The boy look at the game.
To: The boys look at the game.

\d **d** is a single digit. This symbol in **string** is replaced by the set of characters in the input lines that matches the **d**th substring in **pattern**. Substrings begin with the characters **\(** and end with the characters **\)**. For example, the command:

```
s/\(stu\)\(dy\)/\1r\2/
```

tells **sed** to find a pattern **study** in the input line, and copy that pattern to the output with an **r** inserted in the middle. Therefore, it changes the input line:

From: The study chair
To: The sturdy chair

The letters that appear as flags change the replacement as follows,

g Substitute **string** for all instances of **pattern** in the indicated line(s). Characters in **string** are not scanned for a match of **pattern** after they are inserted. For example, the command:

```
s/r/R/
```

changes:
From: the round rock
To: the Round rock

But, the command:

```
s/r/R/g
```

changes:
From: the round rock
To: the Round Rock

p Print (to **stdout**) the line that contains a successfully matched **pattern**.

w filename Write to **filename** the line that contains a successfully matched **pattern**. If **filename** exists, it is overwritten; otherwise, it is created. A maximum of 10 different files can be mentioned as input or output files in the entire editing process. Include exactly one space between **w** and **filename**.

16.0 Chapter 16. Using the Macro Processor (m4)

Subtopics

16.1 CONTENTS

16.2 About This Chapter

This chapter describes how to use the **m4** macro processor, which is a front-end processor for a compiled or assembled programming language. It also contains information on defining macros and using predefined macros.

16.3 The Macro Processor

The **m4** macro processor is a front-end processor for a compiled (or assembled) programming language. The **#define** statement in C language is an example of the facility provided by the macro processor.

At the beginning of a program, you can define a symbolic name or symbolic constant as a particular string of characters. The **m4** macro processor then replaces later unquoted occurrences of the symbolic name with the corresponding string. Besides replacing one string of text with another, the **m4** macro processor provides the following features:

Arithmetic capabilitie
File manipulatio
Conditional macro expansio
String and substring functions

A ***token*** is a string of letters and digits. The **m4** program reads each alphanumeric token and determines if the token is the name of a macro. It then replaces the name of the macro with its defining text, and pushes the resulting string back onto the input to be rescanned. You can call macros with arguments, in which case the arguments are collected and substituted into the right places in the defining text before the defining text is rescanned.

The **m4** program provides built-in macros; you can also define new macros. Built-in and user-defined macros work the same way except that some of the built-in macros change the state of the process. Refer to "Using Other m4 Macros" in topic 16.6 for a list of the macros.

16.4 Using the Macro Preprocessor

To use the **m4** macro processor, enter the following command:

```
m4 [file]
```

The **m4** program processes each argument in order. If there are no arguments or if an argument is **-**, **m4** reads standard input as its input file. The **m4** program writes its results to standard output. Therefore, to redirect the output to a file for later use, use a command like:

```
m4 [file] >outputfile
```

16.5 Defining Macros

The **define** macro is a built-in function that defines macros. For example, if the following statement is in a program:

```
define(name, stuff)
```

the **m4** program defines the string **name** as **stuff**. When the string **name** occurs in a program file, **m4** replaces it with the string **stuff**. The string **name** must be ASCII alphanumeric and must begin with a letter or underscore. The string **stuff** is any text, but if the text contains parentheses the number of open, or left, parentheses must equal the number of close, or right, parentheses.

Use the / (slash) character to spread the text for **stuff** over multiple lines. The open parenthesis must immediately follow the word **define**. For example:

```
define(N, 100)
...
if (i > N)
```

defines **N** to be 100 and uses the symbolic constant **N** in a later **if** statement. Macro calls in a program have the following form:

```
name(arg1,arg2,...argn)
```

A macro name is only recognized if it is surrounded by nonalphanumerics. Using the following example:

```
define(N, 100)
...
if (NNN > 100)
```

the variable **NNN** is not related to the defined macro **N**.

You can define macros in terms of other names. For example:

```
define(N, 100)
define(M, N)
```

defines both **M** and **N** to be 100. If you later change the definition of **N** and assign it a new value, **M** retains the value of 100, not **N**.

The **m4** macro processor expands macro names into their defining text as soon as possible. The string **N** is replaced by 100. Then the string **M** is also replaced by 100. The overall result is the same as using the following input in the first place:

```
define(M, 100)
```

The order of the definitions can be interchanged as follows:

```
define(M, N)
define(N, 100)
```

Now **M** is defined to be the string **N**, so when the value of **M** is requested later, the result is the value of **N** at that time (because the **M** is replaced by **N**, which is replaced by 100).

Subtopics
16.5.1 Using the Quote Characters
16.5.2 Arguments

16.5.1 Using the Quote Characters

To delay the expansion of the arguments of **define**, enclose them in the ***quote*** characters. If you do not change them, the ***quote*** characters are left and right single quotes (' '). See "Changing the Quote Characters" in topic 16.6.1 to change these characters. Any text surrounded by the quote characters is not expanded immediately, but the quote characters are removed. The value of a quoted string is the string with the quote characters removed. If the input is:

```
define(N, 100)
define(M, 'N')
```

The quote characters around the **N** are removed as the argument is being collected. The result of using quote characters is to define **M** as the string **N**, not 100. The general rule is that **m4** always strips off one level of quote characters whenever it evaluates something. This is true even outside of macros. To make the word **define** appear in the output, enter the word in quote characters, as follows:

```
'define' = 1;
```

Another example of using quote characters is redefining **N**. To redefine **N**, delay the evaluation by putting **N** in quote characters. For example:

```
define(N, 100)
...
define('N', 200)
```

To prevent problems from occurring, quote the first argument of a macro. For example, the following fragment does not redefine **N**:

```
define(N, 100)
...
define(N, 200)
```

The **N** in the second definition is replaced by 100. The result is the same as the following statement:

```
define(100, 200)
```

The **m4** program ignores this statement because it can only define names, not numbers.

16.5.2 Arguments

The simplest form of macro processing is replacing one string by another (fixed) string. However, macros can also have arguments, so that you can use the macro in different places with different results. To indicate where an argument is to be used within the replacement text for a macro (the second argument of its definition), use the symbol **$*n*** to indicate the nth argument. When the macro is used, **m4** replaces the symbol with the value of the indicated argument. For example, the symbol:

```
$2
```

refers to the second argument of a macro. Therefore, if you define a macro called **bump** as:

```
define(bump, $1 = $1 + 1)
```

m4 generates code to increment the first argument by 1. The **bump(x)** statement is equivalent to **x = x + 1**.

A macro can have as many arguments as needed. However, you can access only nine arguments using the **$*n*** symbol ($1 through $9). To access arguments past the ninth argument, use the **shift** macro which drops the first argument and reassigns the remaining arguments to the **$*n*** symbols (second argument to $1, third argument to $2 ... tenth argument to $9). Using the **shift** macro more than once allows access to all arguments used with the macro.

The macro name **$0** returns the name of the macro. Arguments that are not supplied are replaced by null strings, so that you can define a macro that concatenates its arguments like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus:

```
cat(x, y, z)
```

is the same as:

```
xyz
```

Arguments $4 through $9 in this example are null since corresponding arguments were not provided.

The **m4** program discards leading unquoted blanks, tabs, or new lines in arguments, but keeps all other white space. Thus:

```
define(a, b c)
```

defines **a** to be **b c**.

Arguments are separated by commas. Use parentheses to enclose arguments containing commas, so that the comma does not end the argument. For example:

```
define(a, (b,c))
```

has only two arguments. The first argument is **a**. The second is **(b,c)**. To use a comma or single parenthesis, enclose it in quote characters.

16.6 Using Other m4 Macros

The **m4** program provides a set of macros that are already defined. The **define** macro already mentioned is one of them. Figure 16-1 lists each of these macros and provides a brief explanation of its function. The following paragraphs further explain many of the macros and how to use them.

Figure 16-1. m4 Built-in Macros

Macro	**Function**
changecom(*l* , *r*)	Changes the left and right comment characters to the characters represented by *l* and *r*.
changequote(*l* , *r*)	Changes the left and right quote characters to the characters represented by *l* and *r*.
decr(*number*)	Returns the value of *number* - 1.
define(*macroname* , *replacement*)	Defines new macro *macroname* with a value of *replacement*.
defn(*macroname*)	Returns the quoted definition of *macroname*.
divert(*number*)	Changes output stream to *number*.
divnum	Returns the value of the current output stream.
dnl	Delete characters up to and including new-line.
dumpdef('*macroname*'...)	Prints the *macroname* and current definition of named macros.
errprint(*string*)	Prints *string* to the diagnostic output file.
eval(*expression*)	Evaluates *expression* as a 32-bit arithmetic expression.
ifdef('*macroname*' , *arg1* , *arg2*)	If macro *macroname* is defined, returns *arg1*; otherwise, it returns *arg2*.
ifelse(*string1* , *string2* , *arg1* , *arg2*)	If *string1* matches *string2*, returns the value of *arg1*; otherwise, returns the value of *arg2*.
include(*file*)	Returns the contents of the file *file*.
incr(*number*)	Returns the value of *number* + 1.
index(*string1* , *string2*)	Returns the character position in *string1* where *string2* starts (starting with character number 0),

	or -1 if *string1* does not contain *string2*.
len:(*string*)	Returns the number of characters in *string*.
m4exit(*code*)	Exits **m4** with a return code of *code*.
m4wrap(*macroname*)	Runs macro *macroname* at the end of **m4**.
maketemp(*string...XXXXX...string*)	Creates a unique file name by replacing the characters *XXXXX* in the argument string with the current process ID.
popdef(*macroname*)	Removes the definition of *macroname* and then defines *macroname* to be its previous value that was saved with the **pushdef** macro.
pushdef(*macroname , replacement*)	Saves the current definition of *macroname* and then defines *macroname* to be *replacement*.
shift(*parameter list*)	Returns all but the first element of *parameter list* to perform a destructive left shift of the list.
sinclude(*file*)	Returns the contents of the file *file*, but does not report an error if it cannot access *file*.
substr(*string , position , length*)	Returns a substring of *string* that begins at character number *position* and is *length* characters long.
syscmd(*command*)	Executes the system command *command* with no return value.
sysval	Gets the return code from the last use of the **syscmd** macro.
traceoff(*macro list*)	Turns off trace for any macro in *macro list*. If *macro list* is null, turns off all tracing.
traceon(*macroname*)	Turns on trace for macro *macroname*. If *macroname* is null, turns trace on for all macros.
translit(*string , set1 , set2*)	Searches *string* for characters that are in *set1*. If it finds any, changes those characters to corresponding characters in *set2*.
undefine('*macroname*')	Removes the definition of *macroname*.

undivert(*number , number*...)	Appends the contents of the indicated diversion numbers to the current diversion.

Subtopics
16.6.1 Changing the Quote Characters
16.6.2 Removing a Macro Definition
16.6.3 Checking for A Defined Macro
16.6.4 Using Integer Arithmetic
16.6.5 Manipulating Files
16.6.6 Redirecting Output
16.6.7 Using System Programs in A Program
16.6.8 Using Unique File Names
16.6.9 Using Conditional Expressions
16.6.10 Manipulating Strings
16.6.11 Printing

16.6.1 Changing the Quote Characters

Quote characters are normally left and right single quotes (' '). If those characters are not convenient, change the quote characters with the following built-in macro:

```
changequote([,])
```

The built-in **changequote** makes the left and right brackets the new quote characters. To restore the original quote characters, use **changequote** without arguments as follows:

```
changequote
```

16.6.2 Removing a Macro Definition

The **undefine** macro removes the definition of some macro or built-in. For example:

```
undefine('N')
```

The macro removes the definition of **N**. **undefine** can also remove built-ins, as follows:

```
undefine('define')
```

Once you remove a built-in macro, you cannot use the definition of the built-in again.

16.6.3 Checking for A Defined Macro

The built-in **ifdef** determines if a macro is currently defined. The **ifdef** macro permits three arguments. If the first argument is defined, the value of **ifdef** is the second argument. If the first argument is not defined, the value of **ifdef** is the third argument. If there is no third argument, the value of **ifdef** is null. If the first argument is undefined, the value of **ifdef** is the third argument.

16.6.4 Using Integer Arithmetic

The **m4** program provides the following built-in functions for doing arithmetic on integers only:

incr Increments its numeric argument by 1.

decr Decrements its numeric argument by 1.

eval Evaluates an arithmetic expression.

Thus, to define a variable as one more than **N**, use the following:

```
define(N, 100)
define(N1, 'incr(N)')
```

which defines **N1** as one more than the current value of **N**.

The **eval** function can evaluate expressions containing the following operators (listed in decreasing order of precedence):

```
unary +  unary -

** or ^ (exponentiation)

* / % (modulus)

+ -

== != <<= >>=

! (not)

& (and)   && (logical and)

| (or) || (logical or)  ^ (exclusive or)
```

Use parentheses to group operations where needed. All operands of an expression must be numeric. The numeric value of a true relation (like 1 > 0) is 1, and false is 0. The precision in eval is 32 bits.

For example, define **M** to be **2==N+1** using eval as follows:

```
define(N, 3)
define(M, 'eval(2==N+1)')
```

Use quote characters around the text that defines a macro unless the text is very simple.

16.6.5 Manipulating Files

To merge a new file in the input, use the built-in function: **include**. For example:

```
include(filename)
```

This function inserts the contents of *filename* in place of the include command.

A fatal error occurs if the file named in **include** cannot be accessed. To avoid a fatal error, use the alternate form **sinclude**. The built-in **sinclude** (silent include) does not write a message, but continues if the file named cannot be accessed.

16.6.6 Redirecting Output

The output of **m4** can be redirected to temporary files during processing, and the collected material can be output upon command. The **m4** program maintains nine possible temporary files, numbered 1 through 9. If you use the built-in macro:

```
divert(n)
```

The **m4** program writes all output from the program after the **divert** function at the end of temporary file, **n**. To return the output to the display screen, use either the **divert** or **divert(0)** command, which resumes the normal output process.

The **m4** program writes all redirected output to the temporary files in numerical order at the end of processing. The **m4** program discards the output if you redirect the output to a temporary file other than 0 through 9.

To bring back the data from all temporary files in numerical order, use the built-in **undivert**. To bring back selected temporary files in a specified order, use the built-in **undivert** with arguments. When using **undivert**, **m4** discards the temporary files that are recovered and does not search the recovered data for macros.

The value of undivert is *not* the diverted text.

The built-in **divnum** returns the number of the currently active temporary files. If you do not change the output file with the **divert** macro, **m4** puts all output in temporary file 0.

16.6.7 Using System Programs in A Program

You can run any program in the operating system from a program by using the **syscmd** built-in. For example, the following statement runs the **date** program:

```
syscmd(date)
```

16.6.8 Using Unique File Names

Use the built-in **maketemp** to make a unique file name from a program. If this macro receives an argument that contains the string **XXXXX**, it changes the **XXXXX** to the process ID of the current process. For example, for the statement:

```
maketemp(myfileXXXXX)
```

the **m4** program returns a string that is **myfile** concatenated with the process ID. Use this string to name a temporary file.

16.6.9 Using Conditional Expressions

The built-in **ifelse** performs conditional testing. In the simplest form:

```
ifelse(a, b, c, d)
```

compares the two strings **a** and **b**. If **a** and **b** are identical, **ifelse** returns the string **c**. If they are not identical, it returns string **d**. For example, you can define a macro called **compare** to compare two strings and return **yes** if they are the same, or **no** if they are different, as follows:

```
define(compare, 'ifelse($1, $2, yes, no)')
```

The quote characters prevent the evaluation of **ifelse** from occurring too early. If the fourth argument is missing, it is treated as empty.

The built-in **ifelse** can have any number of arguments, and therefore, provides a limited form of multiple path decision capability. For example:

```
ifelse(a, b, c, d, e, f, g)
```

This statement is logically the same as the following fragment:

```
if(a == b) x = c;
else if(d == e) x = f;
else  x = g;
return(x);
```

If the final argument is omitted, the result is null, so:

```
ifelse(a, b, c)
```

is **c** if **a** matches **b**, and null otherwise.

16.6.10 Manipulating Strings

The built-in **len** returns the byte length of the string that makes up its argument. Thus:

```
len(abcdef)
```

is 6, and:

```
len((a,b))
```

is 5.

The built-in **dlen** returns the length of the displayable characters in a string. Characters made up from 2-byte codes are displayed as one character. Thus, if the string contains any 2-byte international character support characters, the results of **dlen** will differ from the results of **len**.

The built-in **substr** provides substrings of strings. Using input, **substr(s,** *i*, *n***)** returns the substring of *s* that starts at the *i*th position (origin zero) and is *n* characters long. If *n* is omitted, the rest of the string is returned. For example, the function:

```
substr('now is the time',1)
```

returns the following string:

```
ow is the time
```

The built-in **index(s1, s2)** returns the index (position) in **s1** where the string **s2** occurs, or -1 if it does not occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration. It has the general form:

```
translit(s, f, t)
```

which modifies **s** by replacing any character found in **f** by the corresponding character of **t**. For example, the function:

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If **t** is shorter than **f**, characters that do not have an entry in **t** are deleted. If **t** is not present at all, characters from **f** are deleted from **s**. So:

```
translit(s, aeiou)
```

deletes vowels from string **s**.

The built-in **dnl** deletes all characters that follow it up to and including the next new line. Use this macro to get rid of empty lines. For example, the function:

```
define(N, 100)
define(M, 200)
define(L, 300)
```

results in a new-line at the end of each line that is not part of the definition. These new-line characters are passed to the output. To get rid of the new-lines, add the built-in **dnl** to each of the lines.

```
define(N, 100) dnl
define(M, 200) dnl
define(L, 300) dnl
```

16.6.11 Printing

The built-in **errprint** writes its arguments on the standard error file. For example:

```
errprint ('error')
```

The built-in **dumpdef** dumps the current names and definitions of items named as arguments. If you do not supply arguments, **dumpdef** prints all current names and definitions. Do not forget to quote the names.

17.0 Chapter 17. Creating an Input Language

Subtopics

17.1 CONTENTS

17.2 About This Chapter

This chapter describes **lex**, a program that generates a program from a set of rules. The **lex** program generates a program, called a lexical analyzer, that analyzes input and breaks it into categories, such as: numbers, letters or operators.

The chapter also describes the **yacc** program. This program generates a program from a set of rules. However, the program that **yacc** generates is a parser. program. A parser is a program that analyzes input, using the categories that the lexical analyzer identified, and determines what to do with the input.

17.3 Writing a Lexical Analyzer Program with lex

The **lex** program helps write a C language program that can receive a character stream input and translate that input into program actions. To use the **lex** program, write a specification file that contains the following parts:

Regular expressions
: Character patterns that the generated lexical analyzer recognizes.

Action statements
: C language program fragments that define how the generated lexical analyzer reacts to regular expressions that it recognizes.

The actual format and logic allowed in this file is discussed in "The lex Specification File" in topic 17.4.

Subtopics
17.3.1 What lex Does
17.3.2 How the Lexical Analyzer Works

17.3.1 What lex Does

Using the information in the specification file, the **lex** program generates a C language program to analyze an input stream according to the specifications. The **lex** program puts the output program in a file called **lex.yy.c**. If the output program recognizes a simple one-word input structure, compile the **lex.yy.c** output file using the command:

```
cc lex.yy.c -ll
```

to get an executable lexical analyzer. However, if the lexical analyzer recognizes more than one-word syntax, create a parser to ensure proper handling of the input (see "Creating a Parser with yacc" in topic 17.12).

The **lex.yy.c** output file can be moved to other systems that have a C compiler that supports the **lex** library functions.

The compiled lexical analyzer performs the following functions:

Reads an input stream of characters

Copies the input stream to an output stream

Breaks the input stream into smaller strings that match the regula expressions in the **lex** specification file.

Executes an action for each regular expression that it recognizes The action(s) are C language program fragments in the **lex** specification file. The action fragments can call actions or subroutines outside of the action fragment.

17.3.2 How the Lexical Analyzer Works

The lexical analyzer that **lex** generates uses an analysis method called a ***deterministic finite-state automaton***. This method provides for a limited number of conditions that the lexical analyzer can exist in, along with the rules that determine what state the lexical analyzer is in.

For a simple example, Figure 17-1 shows a chart of a program that has three states: start, good and bad. The program gets a stream of characters for input. It begins in the **start** condition. When it receives the first character, the program compares the character with the rule. If the character is alphabetic (according to the rule), the program changes to the **good** state; if it is not alphabetic, the program changes to the **bad** state. The program stays in **good** until it finds a character that does not match its conditions, and then it moves to **bad** (which terminates the program).

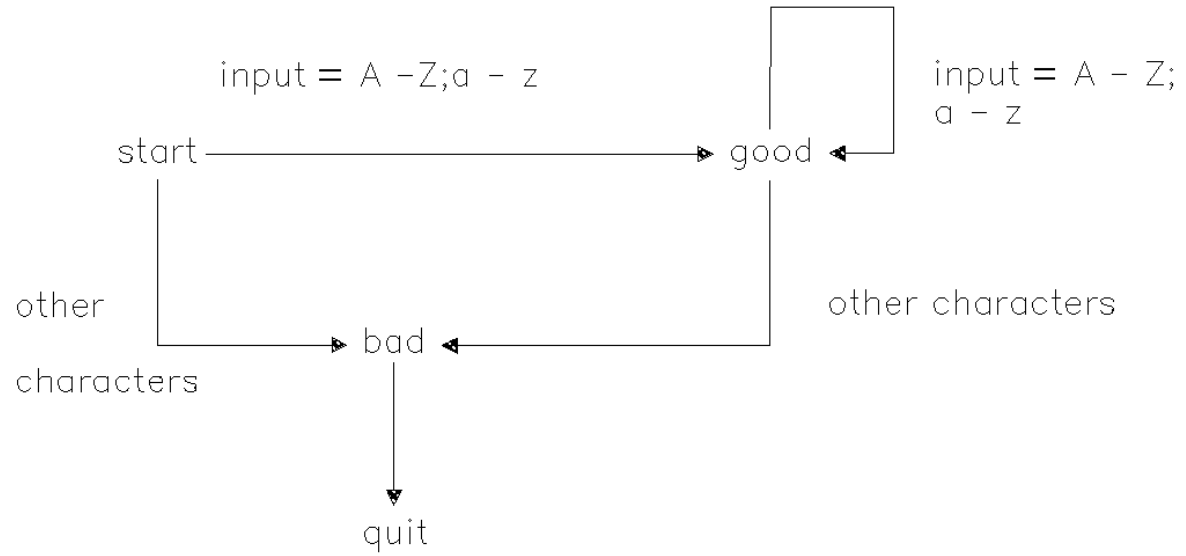


Figure 17-1. Simple Finite State Model

The automaton allows the generated lexical analyzer to look ahead in an input stream more than one or two characters. For example, define two rules in the **lex** specification file, one that looks for the string **ab** and the other that looks for the string **abcdefg**. If the lexical analyzer gets an input string of **abcdefh**, it reads characters to the end of string **abcdefg** before finding that the input string does not match the string **abcdefg**. The lexical analyzer then returns to the rule that looks for the string **ab**, decides that it matches part of the input, and begins trying to find another match using the remaining input **cdefh**.

17.4 The lex Specification File

The format of the **lex** specification file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

You can omit the definitions and the user subroutines. The second **%%** is optional, but the first **%%** is required to mark the beginning of the rules. The minimum **lex** specification file contains no definitions and no rules:

```
% %
```

Without a specified action for a pattern match, the lexical analyzer copies the input pattern to the output without changing it. Therefore, the previous specification file results in a lexical analyzer that copies all input to the output unchanged.

Subtopics
17.4.1 Rules

17.4.1 Rules

The rules section of the specification file contains control decisions that define the lexical analyzer that **lex** generates. The rules are in the form of a table. The left column of the table contains regular expressions; the right column of the table contains ***actions***. Actions are C language program fragments. When the lexical analyzer finds a match for the regular expression that appears in the left column of the table, the lexical analyzer executes the action.

For example, to create a lexical analyzer to look for the string **integer** and print a message when the lexical analyzer finds the string, define a rule:

```
integer      printf("found keyword int");
```

This example uses the C language library function **printf** to print the string. The first blank or tab character in the action indicates the end of the expression. When using only one expression in an action, put it on the same line and to the right of the regular expression (**integer**). When using more than one statement, or if the statement takes more than one line, enclose the action in braces, the same as in a C language program.

For another example, a lexical analyzer to change some words in a file from British spelling to the American spelling has a specification file that contains rules such as:

```
colour                          printf("color");
mechanise                       printf("mechanize");
petrol                          printf("gas");
```

This specification file is not complete because it changes the word **petroleum** to **gaseum**.

17.5 Regular Expressions

Specifying ***regular expressions*** in a **lex** specification file is similar to methods used in **sed** or **ed**. A regular expression specifies a set of strings to be matched. It contains text characters and operator characters. Text characters match the corresponding characters in the strings being compared. Operator characters specify repetitions, choices, and other features.

The letters of the alphabet and the digits are always text characters. For example, the regular expression **integer** matches the string **integer**, and the expression **a57D** looks for the string **a57D**.

Subtopics

17.5.1 Operators

The operator characters for specifying a regular expression are:

Figure 17-2. Regular Expression Operators

Symbol	**Use**
"	Encloses literal strings to interpret as text characters.
\	(Escape character). Indicates that the operator symbol represents the character rather than the operator when used before one of the character class operators. For example, **[\^abc]** represents the class of characters that includes the characters **^abc**.
[ ]	Encloses character classes.
^	In a character class, indicates the complement of the set of characters when the **^** is the first character in a set of characters. For example, **[^abc]** matches all characters except **a**, **b** or **c**, including all special or control characters. Similarly, **[^a-zA-Z]** is any character that is not a letter. In an expression, indicates a match only when the expression is at the beginning of the line when the **^** is the first character in an expression.
0.-	In a character class, indicates a range of characters from the ASCII value of the character that comes before the - to the ASCII value of the character that follows the -. For example, **[a-z0-9]** indicates the character class containing all the lowercase letters and the digits. A range can be either in ascending or descending order, but the order is that of the ASCII values of the characters for AIX. If the program is moved to a system that uses a different set of character codes (like EBCDIC), the range may be a different set of characters. **lex** displays a warning message if moving to another system is likely to cause a problem.
?	(Optional element). Indicates that the character that precedes the **?** is not required to match the string, but may be present in that position. For example, **ab?c** matches either **ac** or **abc**
.	Matches any single character except new-line.
*	Matches any number of consecutive occurrences, including zero, of the character that comes before the *. For example, **a*** is any number of consecutive **a** characters, including zero. The usefulness of matching zero occurrences is more obvious in complicated expressions. For example, the expression, **[A-Za-z][A-Za-z0-9]*** indicates all alphanumeric strings with a leading alphabetic character, including strings that are only one alphabetic character. Use this expression for recognizing identifiers in computer languages.
+	Matches any number of consecutive occurrences, but not zero, of the character that comes before the +. For example, **a+** is one or more instances of **a**. Also, **[a-z]+** is all strings of

lowercase letters.

| | Indicates a match for either the expression that precedes the | or the expression that follows the |. For example, **ab|cd** matches either **ab** or **cd**.

( ) Groups more complex expressions. For example, **(ab|cd+)?(ef)*** matches such strings as **abefef**, **efefef**, **cdef**, or **cddd**; but not **abc**, **abcd**, or **abcdef**.

a/b **/** indicates a match of expression **a** only if expression **b** immediately follows expression **a**. For example, **ab/cd** matches the string **ab** but only if followed by **cd**.

$ Indicates a match only when the expression is at the end of the line when used as the last character in an expression. For example, **ab$** is the same as **ab/\n** where \n is a new-line character. See the description of the **a/b** operator.

{ } When enclosing numbers, the numbers indicate a number of consecutive occurrences of the expression that comes before it. For example, **a{1,5}** indicates a match for from 1 to 5 occurrences of the letter **a**.

When enclosing a name, the name represents a string defined earlier in the specification file. Define the named string in the first part of the **lex** specification, before the rules. For example, **{digit}** looks for a defined string named **digit** and inserts it at that point in the expression.

<x> Encloses a start condition (see "Start Conditions" in topic 17.9). The lexical analyzer executes the associated action only if the lexical analyzer is in the indicated start condition (**x**). If the condition of *being at the beginning of a line* is start condition **ONE**, then the ^ operator would be the same as the expression, **<ONE>**.

To use the operator characters as text characters, indicate that they are text characters by using one of the escape sequences: **" "** (quotes) or \ (backslash). The operator " (quotation mark) indicates that what is between a pair of quotes is text. Thus:

```
xyz"++"
```

matches the string **xyz++**. Note that a part of a string may be quoted. Quoting an ordinary text character has no effect. For example, the expression:

```
"xyz++"
```

is the same as the previous one. Quoting all characters that are not letters or numbers, ensures that text is interpreted as text.

Another way to turn an operator character into a text character is to put a backslash character before it. For example:

```
xyz\+\+
```

```
is another form of the above expressions.
```

17.5.2 Putting Blanks in an Expression

Normally, blanks or tabs end a rule and therefore, the expression that defines a rule. However, you can enclose the blanks or tab characters in quotation marks to include them in the expression. Use quotes around all blanks in expressions that are not already within sets of brackets ([ ]).

17.5.3 Other Special Characters

The **lex** program recognizes many of the normal C language special characters. These character sequences are:

Figure 17-3. Special Characters

Sequence	**Meaning**
\n	New-line - Do not use the actual new-line character in an expression.
\t	Tab
\b	Backspace
\\	Backslash

When using these special characters in an expression, you do not need to enclose them in quotes. Every character, except these special characters and the previously described operator symbols, is always a text character.

17.5.4 Character Classes

Character classes are ranges of characters that **lex** uses to match a single character in the input stream. For example, a character class may contain the letters **a, b, c**. If this character class is a match pattern, **lex** accepts any one of the characters **a, b** or **c** from the input stream.

Define character classes using the [ ] operator pair. Therefore, to define the above character class use the following expression:

```
[abc]
```

The operator symbols, **-**, **^** and **** can also help define the patterns represented in a character class. See "Operators" in topic 17.5.1 for the definitions of these symbols. All other operators within square brackets do not have any meaning other than as an ordinary character.

17.5.5 Matching Rules

When more than one expression can match the current input, **lex** chooses in the following order:

1. The longest match.

2. Among rules that match the same number of characters, the rule that occurs first.

For example, if the rules

```
integer             keyword action...;
[a-z]+             identifier action...;
```

are given in that order, and **integers** is the input word, **lex** matches the input as an identifier, because **[a-z]+** matches eight characters while **integer** matches only seven.

However, if the input is **integer**, both rules match seven characters. **lex** selects the keyword rule because it occurs first. A shorter input, such as **int**, does not match the expression **integer** and so **lex** selects the identifier rule.

Subtopics
17.5.5.1 Matching a String Using Wildcard Characters
17.5.5.2 Finding Strings within Strings

17.5.5.1 Matching a String Using Wildcard Characters

Because **lex** chooses the longest match first, do not use rules containing expressions like .*. For example:

```
'.*'
```

might seem like a good way to recognize a string in single quotes. However, the lexical analyzer reads far ahead, looking for a distant single quote to complete the long match. If a lexical analyzer with such a rule gets the following input:

```
'first' quoted string here, 'second' here
```

it matches:

```
'first' quoted string here, 'second'
```

To find the smaller strings, **first** and **second**, use the following rule:

```
'[^'\n]*'
```

This rule stops after **'first'**.

Errors of this type are not far reaching, because the . (period) operator does not match a new-line character. Therefore, expressions like **.*** stop on the current line. Do not try to defeat this with expressions like **[.\n] +**. The lexical analyzer tries to read the entire input file and an internal buffer overflow occurs.

17.5.5.2 Finding Strings within Strings

The **lex** program partitions the input stream, and does not search for all possible matches of each expression. Each character is accounted for once and only once. For example, to count occurrences of both **she** and **he** in an input text, try the following rules:

```
she                 s++
he                  h++
\n                  |
.                   ;
```

where the last two rules ignore everything besides **he** and **she**. However, because **she** includes **he**, **lex** does *not* recognize the instances of **he** that are included in **she**.

To override this choice, use the action **REJECT**. This directive tells **lex** to go to the next rule. **lex** then adjusts the position of the input pointer to where it was before the first rule was executed, and executes the second choice rule. For example, to count the included instances of **he**, use the following rules:

```
she                 {s++; REJECT;}
he                  {h++; REJECT;}
\n                  |
.                   ;
```

After counting the occurrences of **she**, **lex** rejects the input stream and then counts the occurrences of **he**. Because in this case "she" includes "he" but not vice versa, and you can omit the **REJECT** action on "he". In other cases, it may be difficult to determine which input characters are in both classes.

In general, **REJECT** is useful whenever the purpose of **lex** is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other.

17.6 Actions

When the lexical analyzer matches one of the regular expressions in the rules section of the specification file, it executes the ***action*** that corresponds to the regular expression. Without rules to match all strings in the input stream, the lexical analyzer copies the input to standard output. Therefore, do not create a rule that only copies the input to the output. Use this default output to find gaps in the rules.

When using **lex** to process input for a parser that **yacc** produces, provide rules to match all input strings. Those rules must generate output that **yacc** can interpret.

Subtopics

17.6.1 Null Action

To ignore the input associated with a regular expression, use a ; (C language null statement) as an action. For example:

```
  [ \t\n]  ;
```

ignores the three spacing characters (blank, tab, and new-line).

17.6.2 Same as Next Action

To avoid repeatedly writing the same action, use the **|** (vertical bar) character . This character indicates that the action for this rule is the same as the action for the next rule. For example, the example to ignore blank, tab and new-line characters (shown above), can be written as:

```
  "  "                    |
  "\t"                    |
  "\n"                     ;
```

The quotes around **\n** and **\t** are not required.

17.6.3 Printing a Matched String

To find out what text matched an expression in the rules section of the specification file, include a C language **printf** function as one of the actions for that expression. When the lexical analyzer finds a match in the input stream, the program puts that matched string in an external character array, called **yytext**. To print the matched string, use a rule like:

```
  [a-z]+            printf("%s",yytext);
```

The C language function **printf** accepts a format argument and data to be printed. In this example the arguments to **printf** have the following meanings:

%s A symbol that converts the data to type **string** before printing.
yytext The name of the array containing the data to be printed.

Printing the output like this is common. You may want to define it as a macro in the definitions section of the specification file. If this action is defined as **ECHO**, then the rules section entry looks like:

```
  [a-z]+            ECHO;
```

17.6.4 Finding the Length of a Matched String

To find the number of characters that the lexical analyzer matched for a particular regular expression, use the external variable **yyleng**. For example, to count both the number of words and the number of characters in words in the input, use the following action:

```
[a-zA-Z]+       {words++;chars += yyleng;}
```

This action totals the number of characters in the words matched and puts that number in **chars**.

The following expression finds the last character in the string matched:

```
yytext[yyleng-1]
```

17.6.5 Getting More Input

The lexical analyzer may run out of input before it completely matches an expression in a rules file. In this case, include a call to the **lex** function **yymore** in the action for that rule. Normally, the next string from the input stream overwrites the current entry in **yytext**. If you use **yymore**, the next string from the input stream is added to the end of the current entry in **yytext**.

For example, to define a language that includes the following syntax:

A string is any set of characters between **"** (quotes).

A **** (backslash) must come before all strings.

use the rules:

```
  \"[^"]*   {
                        if (yytext[yyleng-l] =='\\')
                        yymore();
            else
                        ... normal user processing
                        }
```

When this lexical analyzer receives a string such as **"abc\"def"**, it first matches the five characters **"abc**. Then the call to **yymore** adds the next part of the string **"def** to the end. The part of the action code labeled **normal processing** must process the final quote that ends the string.

17.6.6 Putting Characters Back

The lexical analyzer may not need all of the characters that are matched by the currently successful expression, or it may need to return matched characters to the input stream to be checked again for another match. To return characters to the input stream, use the call:

```
yyless(n)
```

where **n** is the number of characters of the current string to keep. Characters that are beyond the **nth** character in the stream are returned to the input stream. This function provides the same type of look ahead that the **/** operator uses, but it allows more control over its usage.

Use the **yyless** function to process text more than once. For example, when parsing a C language program an expression such as **x=-a** is difficult to understand. Does it mean *x is equal to minus a*, or is it an older representation of **x -= a** which means *decrement x by the value of a*? To treat this expression as *x is equal to minus a*, but print a warning message, use a rule such as:

```
=-[a-zA-Z] {
          printf("Operator (=-) ambiguous\n");
          yyless(yyleng-1);
          ... action for =-...
          }
```

17.6.7 Input/Output Routines

The **lex** program allows a program to use the input/output (I/O) routines it uses. These routines are:

input Returns the next input character.

output(c) Writes the character **c** on the output.

unput(c) Pushes the character **c** back onto the input stream to be read later by **input**.

lex provides these routines as macro definitions. You can override them and provide other versions.

These routines define the relationship between external files and internal characters. If you change them, change them all in the same way and they should follow these rules:

All routines must use the same character set

The **input** routine must return a value of zero to indicate end of file.

Do not change the relationship of **unput** to **input** or the look ahead functions will not work.

The standard **lex** library allows the lexical analyzer to back up a maximum of 100 characters.

Create a different version of **input** to be able to read a file containing nulls. Using the normal version of **input**, the returned value of 0 (from the null characters) indicates the end of file and ends the input.

17.6.8 Character Set

The lexical analyzers that **lex** generates process character I/O through the routines **input, output,** and **unput**. Therefore, to return values in **yytext**, **lex** uses the character representation that these routines use. Internally however, **lex** represents each character with a small integer. When using the standard library, this integer is the value of the bit pattern that the computer uses to represent the character. Normally, the letter **a** is represented in the same form as the character constant **'a'**. If you change this interpretation with different I/O routines, put a translation table in the definitions section of the specification file. The translation table begins and ends with lines that contain only the entries:

```
%T
```

The translation table contains lines of the form:

```
%T
{integer} {character string}
{integer} {character string}
{integer} {character string}
%T
```

that indicate the value associated with each character.

17.6.9 End of File Processing

When the lexical analyzer reaches the end of a file, it calls a library routine called **yywrap**. This routine returns a value of 1 to indicate to the lexical analyzer that it should continue with normal wrap-up at the end of input. However, if the lexical analyzer receives input from more than one source, change the **yywrap** function. The new function must get the new input and return a value of 0 to the lexical analyzer. A return value of 0 indicates that the program should continue processing.

You can also include code to print summary reports and tables when the lexical analyzer ends in a new version of **yywrap**. The **yywrap** function is the only way to force **yylex** to recognize the end of input.

17.7 Passing Code to the Generated Program

You can define variables in either the definitions section or the rules section of the specification file. **lex** changes statements in the specification file into a lexical analyzer. Any line in the specification file that **lex** cannot interpret is passed, unchanged, to the lexical analyzer. Three types of entries can be passed to the lexical analyzer in this manner:

Lines beginning with a blank or tab that are not a part of a **lex** rule are copied into the lexical analyzer. If this entry occurs before the first **%%** in the specification file, the entry is external to any function in the code. If the entry occurs after the first **%%**, it must be a C language program fragment that defines a variable. Define these statements before the first **lex** rule in the specification file.

Lines beginning with a blank or tab that are program comments are included as comments in the generated lexical analyzer. The comments must be in the C language format for comments.

Any lines that lie between lines containing only **%{** *and* **%}** is copied to the lexical analyzer. The symbols **%{** and **%}** are not copied. Use this format to enter preprocessor statements that must begin in column 1, or to copy lines that do not look like program statements.

Any lines occurring after the third %% delimiter are copied to the lexical analyzer without format restrictions.

17.8 Defining Substitution Strings

You can define string macros that **lex** expands when it generates the lexical analyzer. Define them before the first %% delimiter in the **lex** specification file. Any line in this section that begins in column 1 and that does not lie between %{ and %} defines a **lex** substitution string. Substitution string definitions have the general format:

```
name                    translation
```

where **name** and **translation** are separated by a least one blank or tab, and **name** begins with a letter. When **lex** finds the string **name** enclosed in { } (braces) in the rules part of the specification file, it changes **name** to the string defined in **translation** and deletes the braces.

For example, to define the names **D** and **E**, put the following definitions before the first %% delimiter in the specification file:

```
D           [0-9]
E           [DEde][-+]{D}+
```

Then, use these names in the rules section of the specification file to make the rules shorter:

```
{D}+              printf("integer");
{D}+"."{D}*({E})? |
{D}*"."{D}+({E})? |
{D}+{E}           printf("real");
```

You can also include the following items in the definitions section:

- Character set tabl
- List of start condition
- Changes to size of arrays to accommodate larger source programs

17.9 Start Conditions

Any rule may be associated with a start condition. **lex** recognizes that rule only when **lex** is in that start condition. You can change the current start condition at any time.

Define start conditions in the definitions section of the specification file by using a line in the following form:

```
% Start  name1 name2
```

The symbols, **name1** and **name2**, are names that represent conditions. There is no limit to the number of conditions and they can appear in any order. You can also shorten the word **Start** to **s** or **S**.

When using a start condition in the rules section of the specification file, enclose the name of the start condition in **<>** (angle brackets) at the beginning of the rule:

```
<name1> expression
```

defines a rule, *expression* that **lex** recognizes only when **lex** is in start condition **name1**. To put **lex** in a particular start condition, execute the action statement (in the action part of a rule):

```
BEGIN name1;
```

This statement changes the start condition to **name1**. To resume the normal state:

```
BEGIN 0;
```

resets **lex** to its initial condition. A rule can be active in several start conditions. For example:

```
<name1,name2,name3>
```

is a legal prefix. Any rule that does not begin with a start condition is always active.

17.10 Compiling the Lexical Analyzer

Compiling a **lex** program is a two-step process:

1. Use **lex** to change the specification file into a C language program. The resulting program is in the **lex.yy.c** file.

2. Use the **cc -ll** command to compile and link the program with a library of **lex** subroutines. The resulting executable program is in the **a.out** file.

For example, if the **lex** specification file is called **lextest**, enter the following commands:

```
lex lextest
cc lex.yy.c -ll
```

Although the default **lex** I/O routines use the C language standard library, the lexical analyzers that **lex** generates do not. You can include different copies of the **input, output,** and **unput** routines to avoid using the library (see "Input/Output Routines" in topic 17.6.7).

17.11 Using lex with yacc

When used alone, the **lex** program generator makes a lexical analyzer that recognizes simple one-word input or receives statistical input. You can also use **lex** with a parser generator, such as **yacc**. The **yacc** program generates a program, called a parser, that analyzes the construction of more than one word input. This parser program operates well with lexical analyzers that **lex** generates. The **lex** program recognizes only regular expressions and formats them into character packages called ***tokens***.

token The smallest independent unit of meaning as defined by either the parser or the lexical analyzer. A token can contain data, a language keyword, an identifier, or other parts of a language syntax.

yacc produces parsers that recognize many types of grammar with no regard to context. These parsers need a preprocessor to recognize input tokens such as the preprocessor that **lex** produces.

When using **lex** to make a lexical analyzer for a parser, the lexical analyzer (from **lex**) partitions the input stream. The parser (from **yacc**) assigns structure to the resulting pieces. Figure 17-4 shows how the two generated programs work together. You can also use other programs along with the programs generated by either **lex** or **yacc**.

The **yacc** program must have a lexical analyzer named **yylex**, which is what the lexical analyzer from **lex** is named. Normally, the default main program in the **lex** library calls this routine, but if **yacc** is loaded and its main program is used, **yacc** calls **yylex**. In this case, each **lex** rule should end with:

```
return(token);
```

where the appropriate token value is returned.

To find out the names for tokens that **yacc** uses, compile the **lex** output file as part of the **yacc** output file by placing the line:

```
#include "lex.yy.c"
```

in the last section of the **yacc** grammar file. For example, if the grammar file is **good** and the specification file is **better**, the final program is created with the following command sequence:

```
yacc good
lex better
cc y.tab.c  lex.yy.c -ly -ll
```

The **yacc** library (**-ly** in the preceding example) should be loaded before the **lex** library to get a main program that invokes the **yacc** parser. You can generate **lex** and **yacc** programs in either order.

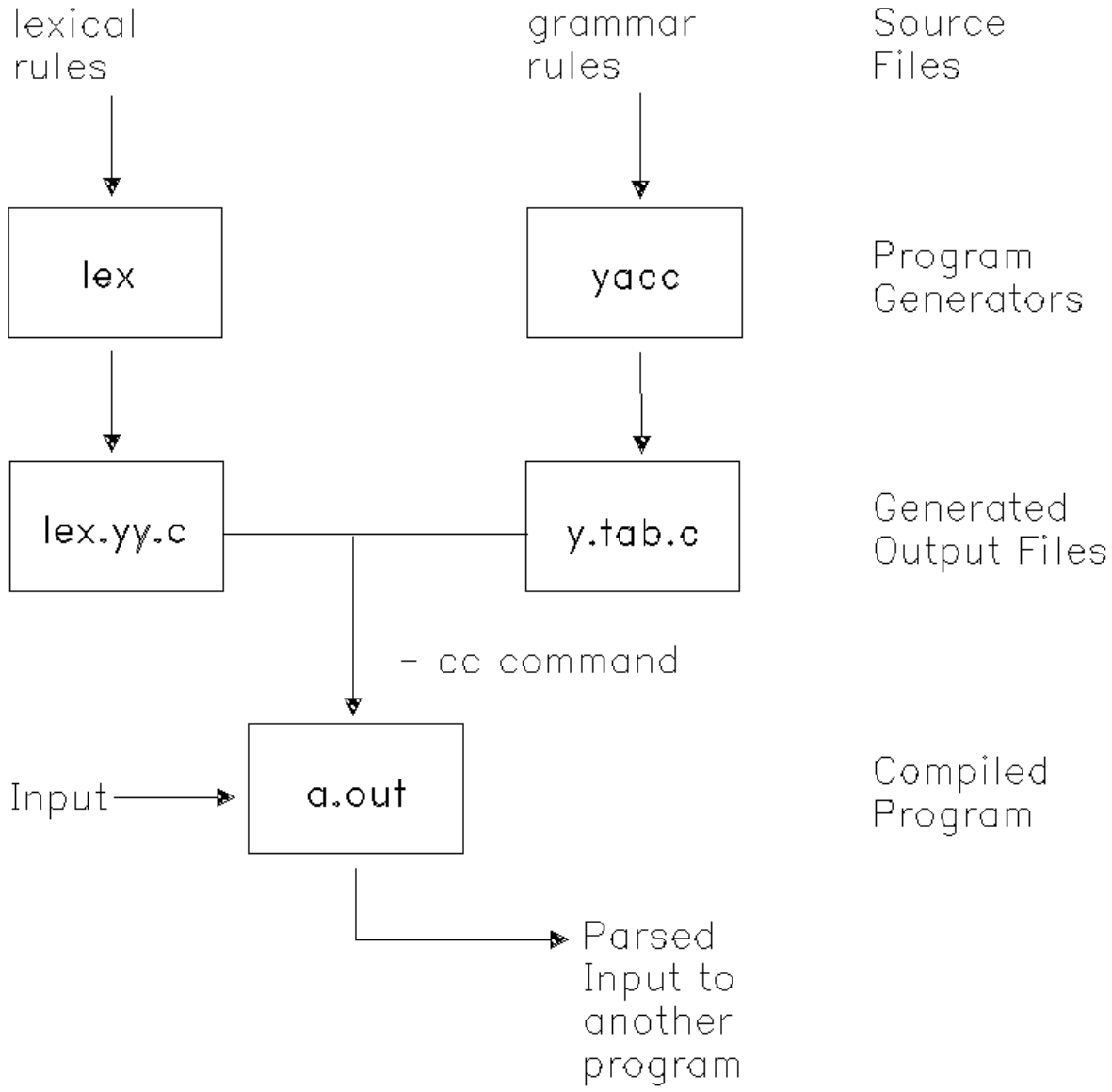


Figure 17-4. lex With yacc

17.12 Creating a Parser with yacc

The **yacc** program creates parsers that define and enforce structure for character input to a computer program. To use this program, supply the following inputs:

Grammar file
: A source file that contains the specifications for the language to recognize. This file also contains the programs **main**, **yyerror** and **yylex**. You must supply these programs.

main
: A C language program that as a minimum contains a call to the function **yyparse** that **yacc** generates. A limited form of this program is in the **yacc** library.

yyerror
: A C language program to handle errors that can occur during parser operation. A limited form of this program is in the **yacc** library.

yylex
: A C language program to perform lexical analysis on the input stream and pass tokens to the parser. You can generate this lexical analyzer using the **lex** program.

When **yacc** gets a specification, it generates a file of C language programs, called **y.tab.c**. When compiled using the **cc** command, these programs form a function called **yyparse** that returns an integer. When it is called, **yyparse** calls **yylex**, the lexical analyzer to get input tokens. **yylex** continues providing input until either the parser detects an error, or **yylex** returns an end-marker token to indicate the end of operation. If an error occurs and **yyparse** cannot recover, it returns a value of 1 to **main**. If it finds the end-marker token, **yyparse** returns a value of 0 to **main**.

17.13 Grammar File

To use **yacc** to generate a parser, give it a grammar file that describes the input data stream and what the parser is to do with the data. The grammar file includes rules describing the input structure, code to be invoked when these rules are recognized, and a routine to do the basic input.

The **yacc** program uses the information in the grammar file to generate a program that controls the input process. This program, called a ***parser***, calls an input routine (the ***lexical analyzer***) to pick up the basic items (called ***tokens***) from the input stream. The parser organizes these tokens according to the structure rules in the grammar file. The structure rules are called ***grammar rules***. When the parser recognizes one of these rules, it executes the user code supplied for that rule. The user code is called an ***action***. Actions return values and use the values returned by other actions.

Use the C programming language to write the action code and other subroutines. **yacc** uses many of the C language syntax conventions for the grammar file.

Subtopics
17.13.1 main and yyerror
17.13.2 yylex

17.13.1 main and yyerror

You must provide these two routines for the parser. To ease the initial effort of using **yacc**, the **yacc** library contains simple versions of **main** and **yyerror**. Include these routines using the **-ly** argument to the loader (or to the **cc** command). The source code for the **main** library program is:

```
main()
{
     return ( yyparse() );
}
```

The source code for the **yyerror** library program is:

```
#include <stdio.h>

yyerror(s)
        char *s;
{
        fprintf( stderr, " %s\n" ,s);
}
```

The argument to **yyerror** is a string containing an error message, usually the string **syntax error**.

These are very limited programs. You should provide more function in these routines. For example, keep track of the input line number and print it along with the message when a syntax error is detected. You may also want to use the value in the external integer variable **yychar**. This variable contains the look-ahead token number at the time the error was detected.

17.13.2 yylex

The input routine that you supply must be able to:

Read the input stream

Recognize basic patterns in the input stream

Pass the patterns to the parser along with tokens that define th pattern to the parser.

A ***token*** is a symbol or name that tells the parser which pattern is being sent to it by the input routine. A ***nonterminal symbol*** is the structure that the parser recognizes.

For example, if the input routine separates an input stream into the tokens of **WORD**, **NUMBER** and **PUNCTUATION**, and it receives the input:

```
I have 9 turkeys.
```

the program could choose to pass the following strings and tokens to the parser:

String	**Token**
I	WORD
have	WORD
9	NUMBER
turkeys	WORD
.	PUNCTUATION

The parser must contain definitions for the tokens that the input routine passes to it. If you use the **-d** option for **yacc**, it generates a list of tokens in a file called **y.tab.h**. This list is a set of **#define** statements that allow the lexical analyzer (**yylex**) to use the same tokens as the parser.

To avoid conflict with the parser, do not use names that begin with the letters **yy**.

You can use **lex** to generate the input routine, or you can write it in the C language. See "The lex Specification File" in topic 17.4 for information about using **lex**.

17.14 Using the Grammar File

A **yacc** grammar file consists of three sections:

    Declaration

    Rule

    Programs

Two **%%** (percent signs) that appear together separate the sections of the grammar file. To make the file easier to read, put the **%%** on a line by themselves. A complete grammar file looks like:

```
declarations
%%
rules
%%
programs
```

The declarations section may be empty. If you omit the programs section, omit the second set of **%%**. Therefore, the smallest **yacc** grammar file is:

```
% %
rules
```

yacc ignores blanks, tabs and new-line characters in the grammar file. Therefore, use these characters to make the grammar file easier to read. Do not, however, use blanks, tabs or new-lines in names or reserved symbols.

Subtopics

17.14.1 Using Comments

Put comments in the grammar file to explain what the program is doing. You can put comments anywhere in the grammar file that you can put a name. However, to make the file easier to read, put the comments on lines by themselves at the beginning of functional blocks of rules. A comment in a **yacc** grammar file looks the same as a comment in a C language program; it is enclosed in **/* */**. For example:

```
/* This is a comment on a line by itself. */
```

17.14.2 Using Literal Strings

A literal string is one or more characters enclosed in '' (single quotes). As in the C language, the \ (backslash) is an escape character within literals, and all the C language escape codes are recognized.

Thus, **yacc** accepts the symbols in the following table:

Figure 17-5. yacc Literal Strings

Symbol	**Definition**
\n	New-line
\r	Return
\'	Single quote (')
\\	Backslash (\)
\t	Tab
\b	Backspace
'\f'	Form feed
'\xxx'	The value **xxx** in octal

Never use **\0** or **0** (the NUL character) in grammar rules.

17.14.3 How to Format the Grammar File

Use the following guidelines to help make the **yacc** grammar file more readable:

Use uppercase letters for token names and lowercase letters fo nonterminal symbol names.

Put grammar rules and actions on separate lines to allow changin either one without changing the other.

Put all rules with the same left side together. Enter the left sid once and use the vertical bar to begin the rest of the rules for that left side.

For each set of rules with the same left side, enter the semicolo once on a line by itself following the last rule for that left side. You can then add new rules easily.

Indent rule bodies by two tab stops and action bodies by three ta stops.

17.14.4 Using Recursion in a Grammar File

Recursion is the process of using a function to define itself. In language definitions, these rules normally take the form:

```
rule        :         <end case>
            |         rule,<end case>
```

which means that the simplest case of the **rule** is the **end case**, but **rule** can also be made up of more than one occurrence of **end case**. The entry in the second line that uses **rule** in the definition of **rule** is the **recursion**. The parser cycles through the input until the stream is reduced to the final **end case**.

When using recursion in a rule, always put the call to the name of the rule as the leftmost entry in the rule (as it is in the above example). If the call to the name of the rule occurs later in the line, such as:

```
rule        :         <end case>
            |         <end case>,rule
```

the parser may run out of internal stack space, stopping the parser.

17.14.5 Errors in the Grammar File

The **yacc** program cannot produce a parser for all sets of grammar specifications. If the grammar rules contradict themselves or require different matching techniques than **yacc** has, **yacc** will not produce a parser. In most cases, **yacc** provides messages to indicate the errors. To correct these errors, redesign the rules in the grammar file, or provide a lexical analyzer (input program to the parser) to recognize the patterns that **yacc** cannot.

17.15 Declarations

The declarations section of the **yacc** grammar file contains:

Declarations for any variables or constants used in other parts of th grammar file.

#include statements to use other files as part of this file (used fo library header files).

Statements that define processing conditions for the generated parser

A declaration for a variable or constant follows the syntax of the C programming language:

type specifier declarator ;

where, **type specifier** is a data type keyword, and *declarator* is the name of the variable or constant. Names can be any length and consist of letters, dots, underscores, and digits. A name cannot begin with a digit. Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

Without declaring a name in the declarations section, you can use that name only as a nonterminal symbol. Define each nonterminal symbol by using it as the left side of at least one rule in the rules section. The **#include** statements are identical to C language syntax, and perform the same function.

The **yacc** program has a set of keywords that define processing conditions for the generated parser. Each of the keywords begin with a **%** and is followed by a list of tokens. These keywords are:

%left Identifies tokens that are left associative with other tokens.

%nonassoc Identifies tokens that are not associative with other tokens.

%right Identifies tokens that are right associative with other tokens.

%start Identifies a name for the start symbol.

%token Identifies the token names that **yacc** accepts. Declare all token names in the declarations section.

All of the tokens on the same line have the same precedence level and associativity; the lines appear in the file in order of increasing precedence or binding strength. Thus:

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. The **+** (plus) and **-** (minus) are left associative and have lower precedence than ***** (asterisk) and **/** (slash), which are also left associative.

Subtopics
17.15.1 Defining Global Variables
17.15.2 Start Conditions

17.15.3 Token Numbers

17.15.1 Defining Global Variables

To define variables to be used by some or all actions, as well as the lexical analyzer, enclose the declarations for those variables in the %{ %} symbols. Declarations enclosed in these symbols are called ***global*** variables. For example, to make the **var** variable available to all parts of the complete program, use the following entry in the declarations section of the grammar file:

```
%{ int var = 0; %}
```

17.15.2 Start Conditions

The parser recognizes a special symbol called the ***start*** symbol. The start symbol is the name of the rule in the rules section of the grammar file that describes the most general structure of the language to be parsed. Because it is the most general structure, it is the structure where the parser starts in its *top down* analysis of the input stream. Declare the start symbol in the declarations section using the keyword, **%start**. If you do not declare the name of the start symbol, the parser uses the name of the first grammar rule in the grammar file.

For example, when parsing a C language procedure, the most general structure for the parser to recognize is:

```
main()
{
   code_segment
}
```

The start symbol should point to the rule that describes this structure. All remaining rules in the file describe ways to identify lower-level structures within the procedure.

17.15.3 Token Numbers

Token numbers are nonnegative integers that represent the names of tokens. If the lexical analyzer passes the token number to the parser instead of the actual token name, both programs must agree on the numbers assigned to the tokens.

You can assign numbers to the tokens used in the **yacc** grammar file. If you do not assign numbers to the tokens, **yacc** assigns numbers using the following rules:

A literal character is the numerical value of the character in th ASCII character set.

Other names are assigned token numbers starting at 257

Note: Do not assign a token number of 0. This number is assigned to the ***endmarker*** token. You cannot redefine it.

To assign a number to a token (including literals) in the declarations section of the grammar file, put a positive integer (not zero) immediately following the token name in the **%token** line. This integer is the token number of the name or literal. Each number must be different from the rest of the token numbers. All lexical analyzers used with **yacc** must return a 0, or a negative value for a token when they reach the end of their input.

17.16 Rules

The ***rules*** section contains one or more grammar rules. Each rule describes a structure and gives it a name. A grammar rule has the form:

```
A : BODY;
```

where **A** is a nonterminal name, and **BODY** is a sequence of zero or more names and literals. The colon and the semicolon are required **yacc** punctuation.

Subtopics
17.16.1 Repeating Nonterminal Names
17.16.2 Empty String
17.16.3 End of Input Marker

17.16.1 Repeating Nonterminal Names

If there are several grammar rules with the same nonterminal name, use the **|** (vertical bar) to avoid rewriting the left side. In addition, use the **;** (semicolon) only at the end of all rules joined by vertical bars. Thus the grammar rules:

```
A  :  B  C  D  ;
A  :  E  F  ;
A  :  G  ;
```

can be given to **yacc** as:

```
A  :  B  C  D
   |  E  F
   |  G
   ;
```

by using the vertical bar.

17.16.2 Empty String

To indicate a nonterminal symbol that matches the empty string, use a **;** (semicolon) by itself in the body of the rule. Therefore, to define a symbol **empty** that matches the empty string, use a rule like the following rule:

```
empty  :  ;
```

17.16.3 End of Input Marker

When the lexical analyzer reaches the end of the input stream, it sends an end of input marker to the parser. This marker is a special token called ***endmarker***, and has a token value of 0. When the parser receives an end of input marker, it checks to see that it has assigned all of the input to defined grammar rules, and that the processed input forms a complete unit (as defined in the **yacc** grammar file). If the input is a complete unit, the parser stops. If the input is not a complete unit, the parser signals an error and stops.

The lexical analyzer must send the end of input marker at the correct time, such as the end of a file, or the end of a record.

17.17 Actions

With each grammar rule, you can specify ***actions*** to be performed each time the parser recognizes the rule in the input stream. Actions return values and obtain the values returned by previous actions. The lexical analyzer can also return values for tokens.

An action is a C language statement that does input and output, calls subprograms, and alters external vectors and variables. Specify an action in the grammar file with one or more statements enclosed in braces **{** and **}**. For example:

```
A  :  '('B')'
   {
     hello(1, "abc" );
   }
```

and

```
XXX  :  YYY  ZZZ
     {
     printf("a message\n");
     flag = 25;
     }
```

are grammar rules with actions.

Subtopics
17.17.1 Passing Values Between Actions
17.17.2 Support for Arbitrary Value Types
17.17.3 Putting Actions in the Middle of Rules

17.17.1 Passing Values Between Actions

An action can get values generated by other actions by using the **yacc** parameter keywords that begin with a dollar sign ($1, $2 ...). The keywords that begin with a dollar sign refer to the values returned by the components of the right side of a rule, reading from left to right. For example, if the rule is:

```
A  :  B  C  D  ;
```

then $2 has the value returned by the rule that recognized C, and $3 the value returned by the rule that recognized D.

To return a value, the action sets the pseudo-variable **$$** to some value. For example, the action:

```
{ $$ = 1;}
```

returns a value of one.

By default, the value of a rule is the value of the first element in it ($1). Therefore, you do not need to provide an action for rules that have the following form:

```
A : B ;
```

17.17.2 Support for Arbitrary Value Types

The values returned by the lexical analyzer and actions are integers. In addition, **yacc** supports values of other types, such as structures. **yacc** also tracks the types and inserts the appropriate union member name so that the resulting parser is type checked. The **yacc** value stack is declared a union of the various types of values desired. The union is declared by the user, and the user associates union member names with each token and nonterminal symbol that has a value. When the **$$** or **$** construction is used to reference a value, **yacc** automatically inserts the appropriate union name so that unwanted conversions do not occur.

The following three methods are used to provide for this typing:

Defining the union

The user must define the union because other subroutines (such as the lexical analyzer) need to know the union member names.

Associating a union member name with tokens and nonterminals

Describing the type of those few values where **yacc** cannot easily determine the type.

To declare the union, you must include the following in the declaration section of your program:

```
%union
 {
        body of union...
 }
```

In the above declaration, the **yacc** value stack and the external variables **yylval** and **yyval** have type equal to the union. If you invoked **yacc** by using the **-d** option, the union declaration is copied onto the **y.tab** file as YYSTYPE.

The union member names must be associated with the various terminal and nonterminal names once YYSTYPE has been defined. The construction

```
<name>
```

indicates a union member name. If the <name> construction follows one of the keywords **%token**, **%right**, and **%nonassoc**, the union member name is associated with the tokens listed. Therefore, if you specify:

```
%left <optype> '+' '-'
```

any reference to values returned by these two tokens is tagged with the union member name **optype**. To associate union member names with nonterminals, the keyword **%type** is used. Therefore, to associate the union member **nodetype** with the nonterminal symbols **expr** and **stat**, you can use the following:

```
%type <nodetype> expr stat
```

17.17.3 Putting Actions in the Middle of Rules

To get control of the parsing process before a rule is completed, write an action in the middle of a rule. If this rule returns a value through the **$** parameters, actions that come after it can use that value. It can use values returned by actions that come before it. Therefore, the rule:

```
A  :  B
         {
            $$ =1;
         }
         C
    {
         x = $2;
         y = $3;
    }
    ;
```

sets x to 1 and y to the value returned by C.

Internally, **yacc** creates a new nonterminal symbol name for the action that occurs in the middle, and it creates a new rule matching this name to the empty string. Therefore, **yacc** treats the above program as if it were written in the following form:

```
$ACT  :  /* empty */
      {
         $$ = 1;
      }
      ;
A     :  B  $ACT  C
      {
         x = $2;
         y = $3;
      }
      ;
```

where $ACT is an empty action.

17.18 Programs

The programs section contains C language programs to perform functions used by the actions in rules section. In addition, if you write a lexical analyzer (input routine to the parser), include it in the programs section.

17.19 Error Handling

When the parser reads an input stream, that input stream might not match the rules in the grammar file. The parser detects the problem as early as possible. If there is an error handling routine in the grammar file, the parser can allow for entering the data again, skipping over the bad data, or a cleanup and recovery action. When the parser finds an error, for example, it may need to reclaim parse tree storage, delete or alter symbol table entries, and set switches to avoid generating any further output.

When an error occurs, the parser stops unless you provide error handling routines. To keep processing the input to find more errors, restart the parser at a point in the input stream where the parser can try to recognize more input. One way to restart the parser when an error occurs is to discard some of the tokens following the error, and try to restart the parser at that point in the input stream.

The **yacc** program has a special token name, **error**, to use for error handling. Put this token in the rules file at places that an input error might occur so that you can provide a recovery routine. If an input error occurs in this position, the parser executes the action for the **error** token, rather than the normal action.

To prevent a single error from producing many error messages, the parser remains in error state until it processes three tokens following an error. If another error occurs while the parser is in the error state, the parser discards the input token and does not produce a message.

As an example, a rule of the form:

```
  stat  :  error ';'
```

tells the parser that, when there is an error, it should skip over the token and all following tokens until it finds the next semicolon. All tokens after the error and before the next semicolon are discarded. When the parser finds the semicolon, it reduces this rule and performs any cleanup action associated with it.

Subtopics
17.19.1 Providing for Error Correcting
17.19.2 Clearing the Look Ahead Token

17.19.1 Providing for Error Correcting

You can also allow the person entering the input stream in an interactive environment to correct any input errors by entering a line in the data stream again. For example:

```
input : error  '\n'
          {
             printf(" Reenter last line: " );
          }
          input
      {
          $$ = $4;
      }
      ;
```

is one way to do this. However, in this example the parser stays in the error state for three input tokens following the error. If the corrected line contains an error in the first three tokens, the parser deletes the tokens and does not give a message. To allow for this condition, use the **yacc** statement:

```
yyerrok;
```

When the parser finds this statement, it leaves the error state and begins processing normally. The error recovery example then becomes:

```
input : error '\n'
          {
             yyerrok;
             printf( "Reenter last line: " );
          }
          input
      {
          $$ = $4
      }
      ;
```

17.19.2 Clearing the Look Ahead Token

The ***look ahead token*** is the next token that the parser examines. When an error occurs, the look ahead token becomes the token at which the error was detected. However, if the error recovery action includes code to find the correct place to start processing again, that code must also change the look ahead token. To clear the look ahead token, include the statement:

```
yyclearin ;
```

in the error recovery action.

17.20 Lexical Analysis

You must provide a lexical analyzer to read the input stream and send tokens (with values, if required) to the parser that **yacc** generates. The lexical analyzer is a function called **yylex**. The function must return an integer that represents the kind of token that was read. The integer is called the ***token number***. In addition, if a value is associated with the token, the lexical analyzer must assign that value to the external variable **yylval**.

To build a lexical analyzer that works well with the parser that **yacc** generates, use the **lex** program (see "The lex Specification File" in topic 17.4).

17.21 Parser Operation

The **yacc** program turns the grammar file into a C language program. That program, when compiled and executed, parses the input according to the grammar specification given.

The parser is a finite state machine with a stack. The parser can read and remember the next input token (called the ***look ahead*** token). The ***current state*** is always the state that is on the top of the stack. The states of the finite state machine are represented by small integers. Initially, the machine is in state 0, the stack contains only 0, and no look ahead token has been read.

The machine can perform one of four actions:

shift n The parser pushes the current state onto the stack, makes **n** the current state, and clears the look ahead token.

reduce r The letter **r** is a rule number. When the parser finds a string defined by rule number **r** in the input stream, the parser replaces that string with the rule number in the output stream.

accept The parser looked at all input, matched it to the grammar specification, and recognized the input as satisfying the highest level structure (defined by the start symbol). This action appears only when the look ahead token is the endmarker and indicates that the parser has successfully done its job.

error The parser cannot continue processing the input stream and still successfully match it with any rule defined in the grammar specification. The input tokens it looked at, together with the look ahead token, cannot be followed by anything that would result in a legal input. The parser reports an error and attempts to recover the situation and resume parsing.

The parser performs the following actions during one process step:

1. Based on its current state, the parser decides whether it needs a look ahead token to decide the action to be taken. If it needs one and does not have one, it calls **yylex** to obtain the next token.

2. Using the current state and the look ahead token if needed, the parser decides on its next action and carries it out. This may result in states being pushed onto the stack or popped off of the stack and in the look ahead token being processed or left alone.

Subtopics
17.21.1 Shift
17.21.2 Reduce

17.21.1 Shift

The **shift** action is the most common action the parser takes. Whenever the parser does a shift, there is always a look ahead token. For example, for the following grammar specification rule:

```
IF shift 34
```

if the parser is in the state that contains this rule and the look ahead token is **IF**, the parser:

1. Pushes the current state down on the stack.
2. Makes state **34** the current state (puts it on the top of the stack).
3. Clears the look ahead token.

17.21.2 Reduce

The ***reduce*** action keeps the stack from growing too large. The parser uses reduce actions after it has matched the right side of a rule with the input stream and is ready to replace the characters in the input stream with the left side of the rule. The parser may have to use the look ahead token to decide if the pattern is a complete match.

Reduce actions are associated with individual grammar rules. Because grammar rules also have small integer numbers, you can easily confuse the meanings of the numbers in the two actions, shift and reduce. For example, the action:

```
. reduce 18
```

refers to grammar rule **18,** while the action:

```
IF shift 34
```

refers to state **34**.

For example, to reduce the rule:

```
A  :  x y z ;
```

The parser pops off the top three states from the stack. The number of states popped equals the number of symbols on the right side of the rule. These states are the ones put on the stack while recognizing **x**, **y**, and **z**. After popping these states, a state is uncovered which is the state the parser was in before beginning to process the rule (the state that needed to recognize rule **A** to satisfy its rule). Using this uncovered state and the symbol on the left side of the rule, the parser performs an action called **goto**, which is similar to a shift of **A**. A new state is obtained, pushed onto the stack, and parsing continues.

The **goto** action is different from an ordinary shift of a token. The look ahead token is cleared by a shift but is not affected by a goto. When the three states are popped, the uncovered state contains an entry such as:

```
A  goto 20
```

causing state **20** to be pushed onto the stack and become the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the parser executes the code that you included in the rule before adjusting the stack. In addition to the stack holding the states, another stack running in parallel with it holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable **yylval** is copied onto the value stack. After executing the code that you provide, the parser performs the reduction. When the parser performs the **goto** action, it copies the external variable **yyval** onto the value stack. The **yacc** variables that begin with **$** refer to the value stack.

17.22 Using Ambiguous Rules

A set of grammar rules is ***ambiguous*** if any possible input string can be structured in two or more different ways. For example, the grammar rule:

```
expr : expr '-' expr
```

states a rule that forms an arithmetic expression by putting two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not specify how to structure all complex inputs. For example, if the input is:

```
expr - expr - expr
```

using that rule, a program could structure this input as either ***left associative***:

```
(  expr - expr ) - expr
```

or as ***right associative***:

```
expr - ( expr - expr )
```

and produce different results.

Subtopics
17.22.1 Understanding Parser Conflicts
17.22.2 How the Parser Responds to Conflicts

17.22.1 Understanding Parser Conflicts

When the parser tries to handle an ambiguous rule, it can become confused over which of its four actions to perform when processing the input. Two major types of conflicts develop:

shift/reduce conflict
: A rule can be evaluated correctly using either a shift action or a reduce action but the result is different.

reduce/reduce conflict
: A rule can be evaluated correctly using one of two different reduce actions, producing two different actions.

A **shift/shift** conflict is not possible. These conflicts result from a rule that is not as complete as it could be. For example, using the previous ambiguous rule, if the parser receives the input:

```
expr - expr - expr
```

after reading the first three parts the parser has:

```
expr - expr
```

which matches the right side of the grammar rule above. The parser can reduce the input by applying this rule. After applying the rule, the input becomes:

```
expr
```

which is the left side of the rule. The parser then reads the final part of the input:

```
- expr
```

and reduces it. This produces a left associative interpretation.

However, the parser can also look ahead in the input stream. If when it receives the first three parts:

```
expr - expr
```

it reads the input stream until it has the next two parts, it then has the following input:

```
expr - expr - expr
```

Applying the rule to the rightmost three parts reduces them to **expr**. The parser then has the expression:

```
expr - expr
```

Reducing the expression once more produces a right associative interpretation.

Therefore, at the point that the parser has read only the first three parts, it can take two legal actions: a shift or a reduction. If the parser has no rule to decide between them, this situation is called a **shift/reduce** conflict.

A similar situation occurs if the parser can choose between two legal reduce actions. That situation is called a **reduce/reduce** conflict.

17.22.2 How the Parser Responds to Conflicts

When shift/reduce or reduce/reduce conflicts occur, **yacc** produces a parser by selecting one of the valid steps wherever it has a choice. If you do not provide a rule that makes the choice, **yacc** uses two rules:

In a shift/reduce conflict, do the shift

In a reduce/reduce conflict, reduce by the grammar rule that it ca apply at the earliest point in the input stream.

Using actions within rules can cause conflicts if the action must be done before the parser can be sure which rule is being recognized. In these cases, using the above rules leads to an incorrect parser. For this reason, **yacc** reports the number of shift/reduce and reduce/reduce conflicts that it resolved using its rules.

17.23 Turning On Debug Mode

For normal operation, the external integer variable **yydebug** is set to 0. However, if you set it to a value that is not zero, the parser generates a description of:

The input tokens that it receive

The actions that it takes for each toke

while it is parsing an input stream.

Set this variable in one of two ways:

Put the C language statement

```
yydebug = 1;
```

in the declarations section of the **yacc** grammar file.

Use **dbx** to execute the final parser, and set the variable on or off using **dbx** commands. See Chapter 12, "Debugging Programs" for information about using **dbx**.

17.24 Creating a Simple Calculator Program - Example

This section describes the example programs for **lex** and **yacc** that are in the set of example programs. The **lex** and **yacc** programs together create a simple desk calculator program that performs addition, subtraction, multiplication and division operations. The calculator program also allows you to assign values to variables (each designated by a single lower case letter) and then use the variables in calculations.

The files that contain the program are:

File	Content
calc.lex	The **lex** specification file that defines the lexical analysis rules.
calc.yacc	The **yacc** grammar file that defines the parsing rules, and calls the **yylex** function created by **lex** to provide input.

To use these files, they must be in your current directory. Copy them from the directory **/usr/lib/samples/toolbook**. The remaining text expects that the current directory is the directory that contains the **lex** and **yacc** example program files.

Subtopics

17.24.1 Compiling the Example Program

Perform the following steps, in order, to create the example program using **lex** and **yacc**:

1. Process the **yacc** grammar file using the **-d** option. The **-d** option tells **yacc** to create a file that defines the tokens it uses in addition to the C language source code:

   ```
   yacc -d calc.yacc
   ```

2. Use the **li** command to verify that the following files were created:

   y.tab.c The C language source file that **yacc** created for the parser.

   y.tab.h A header file containing define statements for the tokens used by the parser.

3. Process the **lex** specification file:

   ```
   lex calc.lex
   ```

4. Use the **li** command to verify that the following file was created:

   lex.yy.c The C language source file that **lex** created for the lexical analyzer.

5. Compile and link the two C language source files:

   ```
   cc y.tab.c lex.yy.c -ly -ll
   ```

6. Use the **li** command to verify that the following files were created:

   y.tab.o The object file for **y.tab.c**.

   lex.yy.o The object file for **lex.yy.c**.

   a.out The executable program file.

You can then run the program directly from **a.out** by entering the command:

```
a.out
```

or, you can move the program to a file with a more descriptive name, like in the following example, and then run it:

```
mv a.out calculate
calculate
```

In either case after you start the program, the cursor moves to the line below the **$** (command prompt); Then enter numbers and operators in calculator fashion. After you press the **Enter** key, the program displays the result of the operation. If you assign a value to a variable:

```
m=4 <enter>
_
```

the cursor moves to the next line. You can then use the variable in calculations and it will have the value assigned to it:

```
m+5 <enter>
9
_
```

17.24.2 The Parser Source Code

Figure 17-6 shows the contents of the file **calc.yacc**. This file has entries in all three of the sections of a **yacc** grammar file: declarations, rules and programs.

```
%{
#include <stdio.h>
int regs[26];
int base;

%}
%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS  /*supplies precedence for unary minus */

%%                   /* beginning of rules section */

list:                        /*empty */
          |
          list stat'\n'
          |
          list error'\n'
          {
            yyerrok;
          }
          ;

stat:     expr
          {
            printf("%d\n",$1);
          }
          |
          LETTER '=' expr
          {
            regs[$1] = $3;
          }
          ;

expr:     '(' expr ')'
          {
            $$ = $2;
          }
          |
          expr '*' expr
          {
            $$ = $1 * $3;
          }
          |
          expr '/' expr
          {
            $$ = $1 / $3;
```

```
          }
          |
          expr '%' expr
          {
            $$ = $1 % $3;
          }
          |
          expr '+' expr
          {
            $$ = $1 + $3;
          }

          |
          expr '-' expr
          {
            $$ = $1 - $3;
          }
          |

          expr '&' expr
          {
            $$ = $1 & $3;
          }
          |
          expr '|' expr
          {
            $$ = $1 | $3;
          }
          |
          '-' expr %prec UMINUS
          {
            $$ = -$2;
          }

          |
          LETTER
          {
            $$ = regs[$1];
          }
          |
          number
          ;

number:   DIGIT
          {
            $$ = $1;
            base = ($1==0) ? 8:10;
          }

          |
          number DIGIT
          {
            $$ = base * $1 + $2;
          }
          ;

%%
main()
{
 return(yyparse());
```

```
  }


  yyerror(s)
  char *s;
  {
    fprintf(stderr," %s\n",s);
  }


  yywrap()
  {
    return(1);
  }
```

Figure 17-6. yacc Grammar File for Calculator Program - calc.yacc

Subtopics
17.24.2.1 Declarations Section
17.24.2.2 Rules Section
17.24.2.3 Programs Section

17.24.2.1 Declarations Section

This section contains entries that perform the following functions:

Includes standard I/O header fil
Defines global variable
Defines the rule **list** as the place to start processing
Defines the tokens used by the parse
Defines the operators and their precedence

17.24.2.2 Rules Section

The rules section defines the rules that parse the input stream.

17.24.2.3 Programs Section

The programs section contains the following routines. Because these routines are included in this file, you do not need to use the **yacc** library when processing this file.

main( )	The required main program that calls **yyparse( )** to start the program.
yyerror(s)	This error handling routine only prints a syntax error message.
yywrap( )	The wrap-up routine that returns a value of 1 when the end of input occurs.

17.24.3 The Lexical Analyzer Source Code

Figure 17-7 shows the contents of the file **calc.lex**. This file contains include statements for standard input and output, as well as for the file **y.tab.h**. The **yacc** program generates that file from the **yacc** grammar file information if you use the **-d** flag with the **yacc** command. The file **y.tab.h** contains definitions for the tokens that the parser program uses. In addition, **calc.lex** contains the rules to generate the tokens from the input stream.

```
%{

#include <stdio.h>
#include "y.tab.h"
int c;
extern int yylval;
%}
%%
" "       ;
[a-z]     {
            c = yytext[0];
            yylval = c - 'a';
            return(LETTER);
          }
[0-9]     {
            c = yytext[0];
            yylval = c - '0';
            return(DIGIT);
          }
[^a-z0-9\b]    {
                 c = yytext[0];
                 return(c);
               }
```

Figure 17-7. lex Specification File for Calculator Program - calc.lex

18.0 Chapter 18. International Character Support

Subtopics

18.1 CONTENTS

18.2 About This Chapter

This chapter includes a discussion of international character support, which pervasively affects character and string handling to support the character sets and strings of a number of languages. Accordingly, international character support is described in terms of all the affected areas, to let you establish and revise environments for character, string, and time and date handling. International character support is provided primarily through subroutines, but the kernel, commands and other utilities are also affected.

18.3 Introduction

The AIX operating system and the programming languages used with it are based on English language characters and on symbols common to the United States of America. Nevertheless, facilities to handle other languages are also provided. This includes character sets and string-handling functions to cover Japanese and many European languages, plus functions to manupulate date, time, and currency formats.

Conversion subroutines are offered to translate between the various character sets and between date and time formats. For the most part, the programmer does not need to be concerned with the details of these manipulations. The programmer merely needs to know which subroutines to call.

Environment variables are used to control the date, time, and currency formats used by each process. Each of these values has a default, and formats may be selected individually or in logical groups. The **LANG** variable, for instance, sets all place- and culture-related variables to the standard formats normally used by the speakers of a particular language. Pointers to tables select collation sequences and the relative values given to characters when they are compared. Conversion subroutines permit passing information to and from other systems which use extended character sets other than those supported by AIX. Other conversion facilities support passing text to attached terminals or remote systems which do not offer extended characters.

18.4 Basic Concepts and Definitions

ASCII vs. extended characters

The foundation of most IBM character sets is the 7-bit **ASCII character set**. These are 128 logical symbols offering digits, punctuation marks, English language letters and a set of non-printing control characters. These characters are always represented by bit values from 0 to 127 (decimal). Any character with a bit value beyond 127 is considered an **extended character**. The PC-DOS and OS/2 operating systems use these extended characters for mathematical and scientific symbols, plus a group of block-based drawing characters.

pc850 and pc932

AIX offers international character support in the form of two extended character sets called *pc850* and *pc932*. *pc850* offers the ASCII characters plus a full set of extended Roman-based characters with diacritical and accent marks. These satisfy the needs of European languages. All *pc850* characters are stored as one byte.

pc932 offers the ASCII characters plus a large complement of extended characters used to write Japanese. *pc932* includes Hiragana, Katakana, Kanji, and double-wide Roman-based characters. This character set contains both single-byte and double-byte characters.

Multibyte characters and their data types

AIX character sets are based on *multibyte* characters. This means that a character may be encoded in any number of bytes. All characters currently used by AIX are stored in files as one or two bytes, but the system uses a four-byte format for fast, accurate collation and comparison of the elements of very large character sets. The same character is often stored in files in a one- or two-byte format (data type **mbchar_t**) and manipulated internally by programs in a four-byte format (data type **wchar_t**).

Code Page

Characters are grouped together in arrays called **Code Pages**. A Code Page which fills the needs of English and other European languages can be defined as 256 unique characters, each of which is represented by a single byte of data. AIX, DOS, and OS/2 all offer such single-byte Code Pages. *pc850* is the Code Page used by AIX for English and other Roman-based European languages. Full details may be found in the *AIX PS/2 Keyboard Description and Character Reference*.

The Japanese language is written with many more symbols than can be represented by a single byte, so AIX extends the normal definition of Code Page to cover *pc932*, an array of over three thousand separate symbols. In this larger sense, a Code Page is an array of any number of code points used to represent a given character set.

Code Point

A **code point** is a conceptual character, it is a bit pattern which corresponds to one single character in a Code Page array. This binary value may correspond to one character in Code Page A and another character in Code Page B. Code points encode both ASCII characters and extended characters.

Each code point in a Code Page represents a single unique character. The same letter may be represented by several code points in the same Code Page (*a* and *A*, for instance), but these are considered different characters.

Characters oriented vs code point oriented

Most programs and operating systems are *character oriented*. They make the assumption that a character is always encoded by a certain number of bits and they will not properly handle characters larger or smaller than that value. AIX Version 1.2.1 is *code point oriented* rather than *character oriented*. AIX processes characters as logical entities which may be expressed in any number of bits needed by the language in use.

Collation concepts

Code points within a Code Page may be divided into **ranges**. The code points within a range may be ordered by a **collating sequence**. This sets up a lexigraphical ("dictionary" or "alphabetical") order for that range. **Equivalence classes** may also be established. This means that certain characters may be considered equal for the purpose of comparison, even though they are expressed by different bit values. An equivalence class is based on the natural grouping of characters for a language or dialect. In English, for instance, *a* is considered equal to *A* for the purpose of a dictionary sort. The two are considered unequal for the purpose of an ASCII sort.

Locale

Each national language group uses its own coordinated set of formats for certain kinds of data. These include the character set used to write the national language, plus distinct formats for the expression of times, dates, numbers, and currency. The set of patterns chosen by each nation is different from the formats used by other language groups.

The format a process uses is indicated by environment variables. Each AIX process is said to run in a specific **locale**, which is the subset of environment variables that establishes these place- and culture-dependent formats.

18.5 History of AIX Character Support

AIX Version 1.0 supported only ASCII characters. Character processing was exclusively single-byte and was character-oriented rather than code-point oriented.

AIX Version 1.2 introduced **National Language Support (NLS)** which supported European languages only. Character processing was still single-byte, but a larger array of characters was available via a Code Page switching system. A number of data types, subroutines, and environment variables were introduced for use with extended Roman-based characters. All were prefixed with **NC** or **NL**. These have been preserved for the sake of backward compatibility.

AIX Version 1.2.1 offers **M***ultibyte* **C***haracter* **S***upport* **(MBCS)** to cover both European-based languages and Japanese. Data types **mbchar_t** and **wchar_t** were introduced to accommodate characters that may be encoded in a variety of sizes and bit patterns, according to the needs of the language. New environment variables were introduced. A large number of the standard string processing routines were modified to process characters as logical entities, irrespective of the number of bytes being used to encode each character.

Note: Programs written under the NLS system will still run under Version 1.2.1, but the NLS system should not be used when writing new programs. Most of the NL and NC entities are now front ends for equivalent MBCS routines. These have been preserved for the sake of backward compatibility only. Calling them in a new program would be inefficient, when the corresponding MBCS entity can be called directly.

18.6 Version 1.2.1 Modifications

The system uses **file code** for storage and transmission of characters. **wide code** (or process code) is used for processing characters within any program that will do more than the simplest character manipulation. The **mbchar_t** data type is used for characters in file code format. The **wchar_t** data type is used to express *wide characters*. AIX Version 1.2.1 offers subroutines to convert back and forth between the two.

The **ctype** macros classify characters by type and the **conv** macros convert from one type to another. (See the *AIX Technical Reference* for details.) All of these macros have been modified to accommodate multibyte characters.

Macros and subroutines throughout the system have been modified to operate on the **wchar_t** character type in ways that are analogous to the processing of the more familiar **char** character type. These modified macros and subroutines are named as their counterparts, with the addition of the prefixes **mb** and **wc**. **mb** indicates a routine that processes character strings which might possibly contain multibyte characters. **wc** indicates a routine that operates on the **wchar_t**, (wide character) format.

Additional macros and subroutines translate between the **wchar_t** and **mbchar_t** character types. **mbtowc** converts the file code formats, **char**s, or **mbchar_t**s, to the internal processing format, **wchar_t**. **wctomb** converts wide characters back to the file code formats.

Dynamically configurable tables are loaded into memory at runtime to support the **wchar_t** data type. One table maps each **wchar_t** to its corresponding **mbchar_t**; another assigns collation weights to each **wchar_t**.

AIX uses system routines to translate times, dates, and currency values into formats appropriate to the nationality selected. The formats produced are controlled by **environment files**, which contain strings of tokens used to modify the operation of these system routines. These environment files are found in **/usr/lib/mbcs/file.en** and there is one for each national language supported by the system. Which of these environment files will be used is dictated by the **LANG** environment variable.

In most cases, the speakers of a given language will always select a coordinated set of formats for their characters, dates, times, and money. **LANG** selects these formats as a logical unit. In the rare instance that a user wants to run a process with some non-standard mixture of formats, other environment variables allow culturally defined formats to be set individually. For further information, see **environment** in the *AIX Operating System Technical Reference*.

AIX offers configurable collation. Character collation is controlled by the dynamically configurable collation tables described above. A number of these are shipped with the system. Each table establishes an alphabetical sorting order for a particular character set. It may also establish character cases and set up equivalence classes. You may alter any of theses for the needs of the dialect used by a particular group, site, or individual. Customized tables are compiled with the **ctab** program and selected with environment variables. For further information, see **ctab** in the *AIX Operating System Commands Reference*.

18.7 Features

AIX subroutines offer the following features for international character support:

Character conversion and collation using code points

Regular expression support for non-ASCII character. A code point i always treated as a single character, even though it may be expressed as more than one byte. Metacharacters retain their traditional interpretation, but now refer to code points rather than particular glyphs.

Revision of character sets, collating sequence, and equivalence clas from an application. A new collating table input file can be selected with the **setlocale** routine.

Extension of all user-visible names by code points. Users may emplo extended characters in file names, including the names of devices, directories, and file systems.

European language support for ASCII terminals. Such terminals do no ordinarily offer extended character support, but AIX supports input and output of code points rather than specific characters. (See "Terminal Maps" in topic 18.9.3.)

Conversion of ASCII text to and from European extended characters The essential characteristics of a file containing pc850 extended characters is preserved.

Support for dialectic date and time formats. All AIX utilities ca output time and date information in the configuration selected. The **at** utility accepts input dates in the currently configured format.

Time zone support. The system can automatically begin and en daylight savings time as specified, using the **TZ** environment variable. See the discussion of **environment** (Miscellaneous Facilities) in the *AIX Operating System Technical Reference*.

Currency and number format support. Selection and placement of th monetary symbol, the thousands divider, and the decimal position character can be varied. (For example, the default for U.S. currency is: $9,999.99).

Conversion between code points and characters. Two subroutines **mbtowc** and **wctomb**, convert from the **char** data type to the **wchar_t** and back again.

Conversion of character strings. Two subroutines, **mbstowcs** and **wcstombs**, convert strings of extended characters to wide character strings, or strings of wide characters to extended character strings, respectively.

Facilities for retrieving the current collation values of code points This information is obtained from the arrays created by **ctab** from a collation table. The **wc_collate** and **wcscoll** subroutines access collation information. **wc_eqvmap** determines equivalence classes.

Facilities for retrieving information on code points. **mblen** determines the number of bytes comprising a character.

String manipulation support. The basic string manipulatio subroutines of AIX have been enhanced to handle strings of extended characters. The **wcstring** and **mbstring** classes are provided to handle extended characters in ways that parallel the handlings given to singlebyte characters by the **string** routines.

Code point extensions to I/O and in-memory format conversio subroutines. The standard conversion subroutines of the general **printf**, **scanf**, **sprintf**, and **sscanf** types have been modified to handle extended characters and strings, both in memory and during I/O.

Conversion between the time formats used internally and those use during input and output. Three subroutines (**NLtmtime**, **NLstrtime**, and **strftime**) convert between a binary representation of time and a text string representation, formatted according to conventions which the user may specify.

Automatic initialization of international character support at AI initialization. The **setlocale()** function is used to initialize the user's locale-dependent variables and formats. They are usually set as a coordinated battery (the date, time, character, and currency expressions used in Switzerland, for instance) according to the dictates of the **LANG** environment variable. For unusual needs, **setlocale()** can also modify the locale settings to any non-standard mixture desired.

18.8 Programming Language Support

AIX does not currently provide direct international character support features for the VS FORTRAN, VS COBOL, or VS Pascal languages. Programs can be written in these languages for purely numerical functions, but none of these languages can process characters of more than one byte directly. Each of these languages is oriented strictly toward singlebyte characters and must not be given multibyte characters to process.

The C library routines, however, offer full support for handling multibyte characters, and AIX allows Version 1.1 of all three of these languages to link with the C libraries. Thus FORTRAN, COBOL, and Pascal programs can gain indirect multibyte character support through the C library routines.

Any such program must first call **setlocale()**. It must do all character processing through the C language library routines. The source code given to these compilers must contain no multibyte characters, and the compilers must not be given filenames or pathnames which contain multibyte characters.

Warning: It is easiest and safest to adhere to the following rule of thumb: Any function that will process characters or accept character input in an environment where users may employ multibyte characters should probably be coded in the C language. VS Pascal, VS COBOL, or VS FORTRAN should not be used for this purpose unless they link to the C libraries and do all character processing through those facilities.

18.9 Kernel Modifications

The AIX kernel has been modified in three major ways: to handle code points, to handle multibyte file names, and to provide terminal map support.

Subtopics
18.9.1 Code Point Support
18.9.2 Multibyte Character File Names
18.9.3 Terminal Maps

18.9.1 Code Point Support

The kernel treats characters as logical entities rather than as specific glyphs. A code point is not assumed to occupy any specific number of bytes.

Tables of mapped characters are provided in the *AIX PS/2 Keyboard Description and Character Reference*. The ordinal ranges of the *pc850* characters can be seen in the "display symbols" listing in the *AIX Operating System Technical Reference*. The two references cited above cover only *pc850*, which provides characters for Roman-based European languages. Similar data for the *pc932* Japanese characters is made available with the Japanese AIX system.

A code point is removed from input with a single backspace character when **ICANON** is set. (See **termio** in the *AIX Operating System Technical Reference*.) This causes a code point to be handled like a character, even if it occupies more than a single byte of storage. Otherwise a single backspace would remove only the last byte of a multibyte character. **ICANON** is set by default.

18.9.2 Multibyte Character File Names

AIX Version 1.2.1 allows file names to be expressed in multibyte extended characters. This includes the names of directories and file systems.

This provision cannot be indiscriminately used, however. Certain files, file systems and directories are always known by their traditional ASCII names (**/etc** and **/bin**, for instance). Renaming these would wreak havoc with system programs.

Multibyte character names should generally be used for user files and user directories only. Any use of multibyte names or entries in system-related files must be avoided.

Other restrictions to the use of extended characters also apply. See "Limits to Support" and "ASCII vs. Multibyte Character Entries" below.

18.9.3 Terminal Maps

AIX provides European language support for a variety of ASCII terminals. This is accomplished through *terminal maps*.

European extended characters are usually ASCII letters with accent marks added. These may be entered at the terminal keyboard as multiple key sequences and sent to the host as a sequence of bytes. They are translated to code points when read by the AIX host. The host may then output a code point, and this is mapped to the sequence of bytes needed by the terminal to form that character on the screen. That sequence of bytes is sent back to the terminal. A variety of terminals are supported in this manner.

AIX supplies terminal maps for many common terminal types and users may create their own. The map provides a list of correspondences between the input sequence which is to be received from the terminal and the code point to which it is mapped. It also records the byte sequence which must be sent to that terminal to produce or display the desired characters.

Many terminals have a hardware capacity to generate, display or emulate European international characters but do not use these features without special provisions. AIX provides additional maps to support these otherwise untapped capabilities.

The usual way to set terminal type is with the **TERM** environment variable. This does not always support the full range of international character support features offered by the terminal. Instead, you can set terminal map values from **/etc/ports**. The two port control parameters used for international character support are **imap**, which sets the file to be used as the terminal's input map and **omap**, which indicates the output map. The **getty** process reads these keywords in **/etc/ports** and loads the indicated terminal maps into memory at the time it initializes the port.

For hardwired ports, the same terminal will always be attached to the same port, usually employed by the same user, or at least by a user with similar language requirements. Therefore the methods above, which always configure the port to support a particular language, are appropriate.

Undedicated ports are always initialized with the default settings, which expect to receive the login and password information in ASCII characters only. Users who log in to undedicated ports (as they do over a modem or an ethernet connection) must start their sessions in ASCII. They are then free to enhance the terminal connection by specifying a set of terminal maps to support their own language.

After a session has begun, the user can change the terminal map by using the **stty** command. (See **stty** in the *AIX Operating System Technical Reference* for full details.)

One place to put such commands is in the user's **.login**, so that the terminal is automatically enhanced for international character support whenever that user logs in.

18.10 Intersystem Compatibility

AIX retains compatibility with environments that do not support international characters. This means AIX can support work stations that do not offer extended characters. Intersystem mail and networking is possible with such systems, and programs may be developed for them on the AIX platform. The same mechanisms that provide translation between environments with and without international character support will also permit interactions between systems that offer different international character support environments.

For the sake of maintaining intersystem compatibility with operating systems that do not support international characters, certain items are always expressed in the ASCII characters which are a subset of every AIX character set. In general, these are items that may be sent to other systems (like login names and passwords) and system files that may be read by other systems (like the stanzas in system attribute files). For details see "Intersystems Compatibility" in *Managing the AIX Operating System*.

Subtopics
18.10.1 Communications with Remote Systems

18.10.1 Communications with Remote Systems

AIX can communicate with other systems via the standard **mail** program and/or via communications protocols such as **uucp**. Communication packages offered with AIX have been modified to support multibyte characters. That same support may or may not be present at the receiving end. The implementations of the same communications packages from other vendors are sometimes limited. The next two sections deal with these issues.

Site names must always conform to the specifications of the communication protocol being used. Since the implementations of **uucp** and other protocols in use by other systems can often handle only ASCII characters, all sitenames must be expressed in ASCII. Some protocols read only a subset of ASCII.

Files may be received from remote systems or transmitted to such systems in file codes other than those used by AIX. AIX uses *pc932* to encode Japanese text, but some other systems use a format called *U-JIS*. Likewise, AIX uses *pc850* to encode European language text, while some other systems use a format called *ISO Latin-1*.

AIX provides a translation utility, **iconv**, to translate one file code into another. Its primary use is to allow speakers of the same language to translate files in that language from one encoding format into another. A Japanese AIX user might compose a message in *pc932* Japanese characters and then use **iconv** to transform it into the *U-J1S* format needed by another Japanese user on a remote system.

It is possible to send messages in extended characters to users on other AIX systems, even if the intervening layers of software read only ASCII characters. The AIX implementation of **uucp**, for instance, passes characters from one system to another without altering them. It is "eight-bit transparent." (See "Interoperability with Other Systems" in Chapter 1 of *Managing the AIX Operating System*.) The command structure given to **uucp** itself, however, must be in ASCII characters.

Discussions of the procedures needed by **uucp** and **mail** are presented below. Use these to illustrate the scope of the support you may be expected to provide for communication between systems.

Subtopics

18.10.1.1 uucp between Systems

The Basic Networking Utilities (**BNU**) is IBM's implementation of the widely used **uucp** package. These commands (**cu** and **uucp**) provide a transparent data path. *cu* provides a data stream for terminal login that will pass multibyte characters accurately. **uucp** will transfer files without distortion to multibyte character contents. For the message to be readable, however, both the sending and the receiving systems must support the same file codes and both sender and receiver must be using the same file code.

The path name and machine name usages, however, are not fully transparent. The name of the host being logged onto or to which the files are being sent must be in ASCII characters only. This is true even if the machine to which the file is being sent is an AIX system which handles multibyte characters. In fact, sitenames in a cluster must always be expressed in ASCII.

AIX allows pathnames to contain multibyte characters, but this should be done with care when using **uucp**. **uucp** composes a *work file* which contains the path names of the source and destination files, the user's login name, the name of the data file in the spool directory, and the user ID to notify on the remote system. The AIX **uucp** software will pass a multibyte character pathname properly. The receiving machine will certainly be able to handle that name if it is part of the same cluster as the sending machine. This is because all machines in a cluster must be running the same version of the operating system, and therefore the implementation of uucp which receives the file will be multibyte capable.

When that work file is read by a non-AIX destination system, or by other systems through which the message may pass, the results may vary. A remote machine running some other operating system or some other version of AIX may be unable to interpret the pathname passed to it. Therefore it is recommended that **uucp** operations be conducted from a directory with a pathname expressed in ASCII characters only. Any pathnames and filenames that will be declared in the work files should be expressed the same way.

In addition, not all vendors' implementations of **uucp** are 8-bit transparent. Multibyte character messages sent to remote systems using a form of **uucp** which employs the eighth bit for any purpose other than character data will scramble file contents.

18.10.1.2 Mail

The standard **mail** program provided with AIX has been modified. It will handle multibyte characters both as message text and as names of files.

This means that the text of **mail** messages sent to users on the same cluster may contain Japanese characters. Incoming Japanese text sent by other users will also be properly handled. The user may write incoming messages to files with Japanese names or include such files in the outgoing **mail** message.

As with all other AIX commands, the standard methodology for use is to invoke the **mail** program by its ASCII name. It is also standard to provide **mail** with its flags and subcommands in single-letter ASCII format.

Japanese text entered into a mail message will be received reliably by another user, provided:

The recipient is running in a Japanese local
The recipient has chosen the same file cod
The recipient is part of the same AIX cluster

Notice that this applies to in-cluster mail only. The transfer is not guaranteed to work reliably between AIX and all other UNIX systems.

Most varieties of UNIX **mail** programs limit messages to seven-bit ASCII. The remote transfer protocols only allow a 7-bit data path. These systems assume that the eighth bit is zero; some even zero out the high-order bit. Such programs will receive Japanese text but be unable to interpret or display it properly. The characters may even be destroyed in the process.

As a general rule, then, Japanese users should feel free to create mail messages in Japanese characters for local users only. **mail** to users on remote systems should contain only ASCII text.

An alternative for **mail** to remote users is the following system:

Each user creates a */mail* subdirectory in his own HOME directory and gives it full "public" permissions. This way his HOME directory and all others can be secured with the standard AIX permissions. All incoming messages will be deposited in this special directory.

The sender creates a Japanese text message with an editor. Thi message should be given an ASCII-character name.

The sender uses either **uucp** or **FTP** to place the message in the recipient's **/mail** directory. If **FTP** is used, the "binary file" option should be employed.

The sender notifies the recipient that the message has been sent vi the standard **mail** program. A **mail** message is sent which simply gives the name of the text file that was transferred.

The recipient retrieves the named file from his **/mail** subdirectory.

18.11 Limits to Support

Some subroutines do not handle **wchar_t** data type; these include essential routines that use the system, control it, or pass characters to or from it. Many of the standard AIX subroutines remain unchanged; strings passed to them must be explicitly converted to an appropriate format with the international character support macros.

File and directory names have a limit of 255 bytes. Code points can be represented by more than one byte of storage, however. Therefore a filename which includes multibyte characters will reach its size limit at some number of characters less than 255. In a case where all the characters used happened to be encoded in two bytes each, the maximum would be 128 characters.

Subtopics
18.11.1 ASCII vs. Multibyte Character Entries

18.11.1 ASCII vs. Multibyte Character Entries

In general, AIX Version 1.2.1 permits almost any interaction which could be handled in ASCII characters to be performed in multibyte characters. There are significant exceptions to this rule, however, and these are noted below.

Items which CAN be expressed in multibyte characters are:

File name
The contents of text file
Path name
Aliases for shell scripts and command names (C Shell users only
Names linked to command or script
Names of new programs specially created for AIX Version 1.2.
Comments and literal strings in a C progra

Items which CANNOT be entered in multibyte characters are:

Sitenames (the names of cluster hosts
Names of remote machine
User login name
User password
Group name
Names of environmental variable
Pathname delimiter
Command names
Command flags
The contents (stanzas) of system attribute files
The coding in a C program (or any other programming language
Telephone numbers and dialer string

All of the above **must** *be entered in ASCII characters only*. The items with asterisks have exceptions, but require considerable care and expertise to accomplish. The safest course is to express these items in ASCII.

18.12 Collation

You can use the **ctab** command to create a new table file which defines the collating sequence and the case conversion of a particular character set. The input and output files are stored in the conventional **/usr/lib/mbcs** directory. Files to support a variety of user environments are supplied with the system, and you can create customized versions for dialects or other special purposes. File names generally reflect their purpose. For example, the files containing the characters, symbols and lexographical collating sequence for Britain have **uk** in their names. Those for the U.S. have **us**.

ctab allows you to use an existing language table file without change, modify an existing file, or create a new one. At system initialization, the **setlocale** subroutine gets an output file and loads it into memory. **setlocale** can also be called from within a program to select a different table file for the use of that program. **nllanginfo** can be used in the same way to return various information about the current locale settings.

For each character in a collating sequence (each **wchar_t**) the table input file provides the following information:

For characters that have case, the corresponding upper- or lowercas version of a given character.

The collating sequence

The characters in each equivalence class. This is used for regula expression processing.

A discussion of setting up collation tables is also contained in "Overview of International Character Support" in *Managing the Operating System*. See **ctab** in the *AIX Operating System Commands Reference* for information on setting **ctype** attributes. See **regexp** in the *AIX Operating System Technical Reference* for detailed information on regular expression programming.

18.13 Modifications to Files and Directories

Some files have been modified and others added to provide international character support. New directories contain groups of files for international character support.

/usr/lib/mbcs/*
Contains character collation tables. Input tables are of the form *ctable*.**ctab**, and output tables are binary arrays of the form *ctable*. If no collation table is set by environment variables, no language tables are used. The values for the **C locale** are used:

8-bit ASCII is recognized; collation is by actual ASCII character value; date, time, and currency formats are set to the traditional UNIX defaults.

/etc/environment
A shell script which provides system-wide environment variables that seldom change (like the timezone).

/etc/profile
A shell script which provides environment variables suitable for most of the users of a system. Individuals may modify these settings automatically in their own **.profile** files.

/$HOME/.profile
Provides user-specific variables.

/usr/include
Holds header files that provide international character support:

1. **/usr/include/mbcs.h** - provides types and prototypes for **mb** and **wc** functions.

2. **/usr/include/NLregexp.h** - provides regular expression support for extended character sets.

3. **/usr/include/ctype.h** - provides support of character type conversion.

4. **/usr/include/NLchar.h** - provides support for the **NLchar** character type.

/usr/include/sys
Some header files in this directory have been modified to provide support for terminal mapping.

1. **termio.h**

2. **ttmap.h**

3. **tty.h**

/usr/nls/termmap/*
Contains terminal maps.

Subtopics
18.13.1 Obtaining Character Information

18.13.1 Obtaining Character Information

The system uses file code formats which represent code points in a variable number of bytes. The *pc850* characters are all a single byte in length, but the *pc932* characters may be stored as one- or two-byte entities, according to which character is being represented. Any application which manipulates characters in file code format needs to know the length of each character in order to handle them properly. Some functions also need to know the display width. Several macros return information about the length of characters expressed in file code:

mblen
: Returns the length in bytes (1 or 2) of the character pointed to by a pointer of the type **mbchar_t**. If the character indicated is not a valid code point, the function returns -1.

wctomb
: This routine converts wide characters to multibyte characters. As an ancillary function, it also returns the number of bytes of the character generated.

mbdwidth
: Returns the display width of a file code character.

NCisxxxx
: A collection of routines which return the type of the character pointed to (Is this character alphabetic? Is it numeric? Is it lowercase? etc.).

Similar facilities are usually available for strings of file code characters. Applications which manipulate wide character data need no such facilities because **wchar_t** is always a uniform four bytes.

For details, see "The C Library" in Chapter 7 of this book.

18.14 cron and /etc/openfiles

Certain system files are used so frequently that it would be a large drain on system resources to open and close them repeatedly. One of the lesser known functions of the **cron** program is to enhance system performance by keeping these files constantly open. The inode remains always in memory.

In previous versions of AIX, **cron** worked from an internal list. Version 1.2.1 uses message catalogs and other language-related files so frequently that these need to be added to the list of files held open. In addition, it is desirable to allow the system administrator to customize this important aspect of system operation.

For these and other performance-related reasons, **cron** now consults an editable text file called **/etc/openfiles**, for the list of files to hold open. **/etc/openfiles** is supplied with the system, but the system administrator may alter it to suit the needs of the installation.

The first ten files listed in **/etc/openfiles** were previously opened by **cron** from its internal list:

```
/bin
/lib
/usr
/usr/bin
/usr/lib
/etc
/tmp
/usr/adm
/
/local
```

These heavily used system directories should never be removed from the list. The three system files which follow should be treated the same way:

```
/etc/passwd
/etc/group
/usr/lib/terminfo
```

Below these entries appear a number of lines which begin with comment symbols. Each names a file which should be held open if a certain program or package is run on that site. These entries should be uncommented according to the needs of the site being configured.

```
#
# Uncomment the next line if TCF is installed
#/etc/site

# Uncomment the next line if TCP/IP is installed
#/etc/hosts

# Uncomment the next line if the nameserver is used
#/etc/resolv.conf
```

Each site will support at least one language. This requires the use of at least one coordinated set of message catalogs. The message catalogs listed in the delivered version of **/etc/openfiles** are those for the default language, U.S. English. If you are configuring the site to use another locale, change the "En.pc850" to the appropriate directory name.

```
/usr/lib/mbcs
/usr/lib/mbcs/msg
/usr/lib/mbcs/msg/En.pc850
/usr/lib/mbcs/msg/En.pc850/system.cat
/usr/lib/mbcs/msg/En.pc850/sh.cat
/usr/lib/mbcs/msg/En.pc850/bas2.cat
/usr/lib/mbcs/msg/En.pc850/lib.cat
/usr/lib/mbcs/msg/En.pc850/ext.cat
```

For locales other than U.S. English, four other language-related files should be added: an environment file, a character table, a collation table, and a conversion table. Every program that deals with text will need to open and close these four.

The entries shown below are presented as examples only. They would be correct for the French language as used in the Swiss locale.

```
# If using a locale other than U.S. English,
# replace the next line with
# the pathname of your environment file.
#/usr/lib/mbcs/Fr_SW.pc850.en

# If using a locale other than U.S. English,
# replace the next line with
# the pathname of your character table file.
#/usr/lib/mbcs/Fr_FR.pc850

# If using a locale other than U.S. English,
# replace the next line with
# the pathname of your conversion table file.
#/usr/lib/mbcs/pc850.cct

# If using a locale other than U.S. English,
# replace the next line with
# the pathname of your collation table file.
#/usr/lib/mbcs/Fr_FR.pc850
```

Other candidates for inclusion would be any software development tools that are heavily used by the programmers on the site. The file which describes the most heavily used terminal at your installation is one more entry to consider. The descriptor file for the VT100, for instance, is **/usr/lib/terminfo/v/vt100**.

One way to determine the most frequently used files on a site is with the **crash** program. The **netlog** subcommand produces a display log of recent net messages:

```
netlog 200 >logfile
```

This subcommand will yield a record of the last 200 net messages. Look for *open* requests in the listing (**grep** for "open"). The results are your candidates.

In the rare instance that users on site or cluster function in several locales, the system administrator may wish to include several sets of language-related files in the list. The language files for the principal non-English language of the site should always be included. Whether those for a secondary non-English language should be included will depend upon how heavily that language is used and upon how many other files are being held open.

Subtopics
18.14.1 Performance Tuning with cron and /etc/openfiles

18.14.1 Performance Tuning with cron and /etc/openfiles

The number of files held open should not be indiscriminately increased. The maximum permitted is 48, but there will come a point where holding files open ceases to help performance and starts to hinder. Like any performance tuning, the best results are obtained by careful testing.

Once you have chosen the files for **cron** to maintain, any extraneous comment lines should be removed from **/etc/openfiles**. **cron** rereads this file periodically; rereading unnecessary comment lines would be wasteful. If you feel that **/etc/openfiles** will need to be modified in the future and the comment lines would be helpful, make a backup copy before erasing these lines.

To provide for files which are removed or recreated, **cron** closes all the open files at regular intervals. Then it rereads **/etc/openfiles** and reopens files based on the contents of **/etc/openfiles**, which may have changed since the time of **cron**'s last reading. The default for this cycle is 20 minutes, but you may arrange to have it done more or less frequently by running **cron** with the **-f** option.

```
cron -f 10
```

This command runs **cron** with a reread interval of 10 minutes. It would normally be placed in **/etc/rc**. This reread interval is another factor that can be adjusted for system tuning. The best interval is determined by trial.

Any changes you make to **/etc/openfiles** will not take effect until the next time **cron** reads the file. If you have set this interval to some long delay and you want your changes to take effect immediately, you may wish to force a reread. This is done by sending **cron** the hangup signal:

```
kill -HUP PID
```

where PID is the Process ID of the **cron** daemon.

18.15 Multibyte Character Set Support

The AIX Operating System provides international character support for both the European languages and the Japanese language. AIX international character support is multibyte character set (MBCS) support, which uses a set of codes (rather than a single code) to represent ASCII and extended characters in files, on networks, and in storage. MBCS support combines (1) ASCII support for English, (2) single-byte national language support (NLS) for the European languages, and (3) double-byte character set (DBCS) support for the Japanese language.

18.16 Programming for an MBCS Environment

Programming for an MBCS environment requires certain specific knowledge. There are specific library routines to be used for character data. Programs ported from previous systems require special handling. The C language offers specific data types to be used for file code characters, and others for wide character data.

18.17 How to Write Codeset-Independent Programs

Subtopics

18.17.1 Character Set Design

There are two ways within MBCS to represent a character. Internal or **process code** is fixed width in order to speed up character operations. External or **file code** is variable width. Typically, it is smaller than process code, making it more convenient to store data. However, it is cumbersome to manipulate in this form since you must always check how many bytes make up a particular character.

The basic paradigm used in MBCS is to use process code if any significant data manipulation is necessary. To facilitate this model, new standard I/O format conversion characters have been added to allow one to read and/or write process code. The translation from/to file code is done before any data is transferred between the device and a program. Thus, the picture seen by the user is that one is actually able to read and write process code.

There are character types for both process and file codes. A character in process code is called a wide character. A wide character is defined as:

```
typedef unsigned long wchar_t
```

A character encoded in file code is called a multibyte character. Since multibyte characters can have variable width, they are typically processed as byte streams, and as such are declared as character arrays. Thus the declaration:

```
char *mbs
```

will be sufficient to access multibyte characters. Occasionally it is necessary to store a character in a integral data type. This can be useful for character comparisons. The data type for a multibyte character is:

```
typedef unsigned long mbchar_t
```

The difference between **mbchar_t** and **wchar_t** is that **mbchar_t** may not necessarily require all 4 bytes for its encoding, whereas a **wchar_t** always takes 4 bytes. A **mbchar** should ONLY be used when dealing with a single character as an atom and will not be the usual case.

All supported file codes will never have a single byte of 0 be part of a multibyte character. So, the null character will be defined as:

```
wide character null:      4 bytes long containing 0.

multibyte character null: 1 byte long containing 0.
```

18.17.2 Codeset Independence

MBCS has been designed in such a way as to minimize the changes to the system and to isolate most of them to the libraries. This facilitates writing application programs in a codeset independent manner. Codeset independence can be achieved by never assuming that a byte and a character are necessarily the same thing. Code that manipulates characters is usually written as:

```
while (*p) {
       /* do some processing on p */
       p++;
}
```

If 'p' is declared as:

```
char *p;
```

then it means that this code will potentially need to handle multibyte characters. Thus 'p' can no longer be advanced using 'p++'. There are routines which handle pointer advancement and are described in several sections below. For now note the following:

```
p++;
```

would become:

```
p = mbsadvance(p);
```

It is obvious that the extra overhead of a function call slows performance if a significant amount of processing is to be performed on the pointer 'p'. Instead, the code could be modified to convert multibyte characters to wide characters or process code. All of the I/O routines support automatic conversion from file code to process code. Other conversion routines such as **mbtowc()** are discussed in detail below.

If the code is designed to handle process code, 'p' will be declared as:

```
wchar_t *p;
```

Since wide characters are a fixed length, you can advance a pointer by 'p++'. However, character and string constants can no longer be written as:

```
'A' or "FOO"
```

This is interpreted by the compiler as multibyte characters (file code) rather than wide characters (process code). Instead, it will be necessary to prefix the character constant by the 'L' character to indicate to the compiler that this is a wide literal:

```
wchar_t wc = L'A';
wchar_t *s = L"FOO";
```

To support programmers' needs, MBCS provides string routines which handle both multibyte and wide characters.

18.17.3 Collation

wc_collate produces the collation value for wide characters. The ordinal value of the characters themselves should never be used for relative comparisons (i.e. <, >, =, =). For these types of comparisons, you should take the collation value of the wide characters (using **wc_collate()**).

wc_collate also provides information for extended collation (an n-to-1 mapping). If the collation value returned by **wc_collate** is negative, the character may possibly make up a larger collation object. In this event, you should pass this value (the negative collation value) to **wc_xcol**, along with the original character and the address of a pointer to that character in the string. **wc_xcol** matches the longest possible collation object in the string, returns the collation value of that object, and advances the string pointer past the object.

Note: This functionality is useful in Latin derived languages.

Latin derived languages also have the notion of "diacriticals" in which the glyphs of various Roman characters are modified to represent different sounds. Although these letters each have unique collating values, in some circumstances it is desirable to map the letters by the Roman letter to which they are equivalent. This equivalence mapping is performed by the **wc_eqvmap()** function.

wc_eqvmap() is a predicate function which returns FALSE (0) if the given **wchar_t** is not the first character of an equivalence class. Using this function (and the fact that equivalence classes are required to contain only consecutive characters) it is possible to determine the beginning and end of a given equivalence class. Since it is possible to create a collation table in which several characters to map to the same collation value, it is necessary to find a unique value for the character before determining its equivalence class. **wc_coluniq()** is the function that finds this unique collating value.

The sequence of code to determine whether a character (**some_char**) falls into the same equivalence class as another character (**other_char**) would be written as follows:

```
test_equivalent(wchar_t some_char,wchar_t other_char)
{
           long some_col, range_start, range_end;

           some_col=wc_coluniq(some_char));
           range_start = range_end = wc_coluniq(other_char);

           while (wc_eqvmap(range_end+1) == 0)
               range_end++;

          while (wc_eqvmap(range_start) == 0)
               range_start--;

         if (range_start <= some_col && some_col <= range_end)
                  return TRUE;
         else
                  return FALSE;
}
```

18.17.4 Input and Output

Much of the work of adapting an application to a particular locale is tailoring the expected input and formatted output to fit the target language. Several features of MBCS support this task, including enhanced **printf/scanf** format strings, **wchar_t** oriented input and output facilities, and message catalogs.

18.17.5 Enhanced printf/scanf Format Strings

printf and **scanf** now allow the 'w' format modifier in format strings to specify wide characters (i.e. **wchar_t**). The appropriate conversion from/to file code to/from process code will be performed by the library call.

18.17.6 wchar_t Oriented Input and Output

The new library routines **getwc()/getwchar()/getws(**) /**fgetws()** and **putwc()/putwchar()/putws()/fputws()** are available in **stdio.h**. These routines are analogous to the traditional **getc()** /**getchar()** and **putc()/putchar()** routines.

18.17.7 Message Catalogs

MBCS supports message encapsulation (the ability to retrieve messages from a catalog) by providing several message catalogs containing all the messages generated throughout the system.

The simplest way to access these message catalogs is to determine the message catalog, set, and message number you wish to display, and then call the **NLgetamsg()** function. **NLgetamsg** returns a pointer to the message in question, or (in the event of an error) returns the default string defined in the call. For example:

```
char * msg = NLgetamsg("fake_compiler.cat",1,3,"Beginning Pass2");
```

retrieves message 3 from set 1 of the catalog 'fake_compiler.cat' The message presumably contains information about the state of the compiler. If the message or catalog is not available, **NLgetamsg** returns the default string (defined in this case to be **"Beginning Pass 2"**).

18.17.8 Non-Positional Format Parameters

In order to make the message catalogs more useful, **printf** and **scanf** allow position specifiers within format specifiers (as required by XPG3). In the following example, the **NLgetamsg()** call demonstrates the use of these position specifiers:

```
char * msg = NLgetamsg("compiler.cat",1,4,"%s: Syntax Error line %d\n");
```

The default message strings demonstrate that there are variable insertions to be performed within the strings before displaying the message, namely a string (the compiler name) and a decimal (the line number in which the syntax error occurred). The display of this message looks like the following:

```
fprintf(stderr, msg, program_name, source_line_number);
```

fprintf would perform the proper variable substitution upon the format string before displaying the message, so that the final result would appear on **stderr** as:

```
fake_compiler: Syntax Error line 23
```

In the course of internationalizing the **fake_compiler** program, the message catalog **compiler.cat** would need to be translated into the language of the target locale.

NLgetamsg() takes into account the current locale (defined by the **setlocale()** function call) when searching for the proper message catalog. The **fake_compiler** program should perform the following function call:

```
setlocale(LC_ALL,"");
```

to initialize the MBCS library functions for the current locale. The **LC_ALL** parameter tells **setlocale()** to initialize all facets of the locale to that locale specified in the second parameter. The second parameter, if a string of zero length, tells **setlocale()** to look up the environment variable **LC_ALL** for the current locale.

Due to the nature of the target language, the translated messages may require that the message be rearranged somewhat. This rearrangement is permitted, and will not break the message, if the message contains position specifiers within the formats. For example, in the **fake_compiler** program there is a message that reads:

```
"%s: line %d Parameter '%s' cannot be of type '%s'"
```

which describes messages of the type:

```
fake_compiler: line 23 Parameter dest cannot be of type 'const int'
```

In translation it may be necessary to reorder the word in the message such that the third parameter (**dest**) now needs to follow the fourth (**const int**). This is possible if the message format patterns contain position specifiers. Examine the original message with position specifiers:

```
"%1$s: line %2$d Parameter '%3$s' cannot be of type '%4$s'"
```

Now there is no ambiguity in the format specifiers, and they can be rearranged freely. For example,

```
  "%1$s: inelay %2$d Ypetay '%4$s' otnay allowedmay orfay arameterpay '%3$s'"
```

will appear as:

```
  fake_compiler: inelay 23 Ypetay 'const int' otnay allowedmay orfay
    arameterpay 'dest'"
```

In both cases, the **printf** call would appear as:

```
  fprintf(stderr,
          NLgetamsg("fake_compiler.cat",SET_TYPES,MSG_NOTALLOWED,
               "%s: line %d Parameter '%s' cannot be of type '%s'"),
          program_name,
          source_line_number,
          parm_name,
          parm_type_name);
```

18.18 Message Catalog Generation

This sections describes the steps to be taken to create, generate, and implement the use of message catalogs for user applications written using the C programming language. It describes the syntax that is used in the generation of a message catalog. There are various commands used to generate the binary message catalog from a message text source file and to output messages within the catalog and the entire catalog to verify its correctness. In addition, it describes the various library routines included in user programs to open, read, and display messages from within a binary catalog.

The AIX Message Facility allows user messages to be separated from program source code. The separated messages may be easily edited or translated to other languages. If the separation of the messages and the subsequent retrieval operation are designed correctly, the same binary image of a program can be executed by different users, each using a different national language set. The person who edits or translates the messages does not require access to the source code; the source code does not need to be recompiled when messages are modified or translated.

Subtopics

18.18.1 The Message Text Source File

A message catalog is a file that contains messages from one or more executable programs. Each message in a message catalog is identified by a message set number and message ID number. These numbers are assigned when the message catalog is created, and are used by the programs accessing the message catalog to specify a particular message to retrieve. Message sets can be used to group a number of related messages together logically within a catalog.

The message catalog is the binary image that is created from the message text source file. The message text source file is initially created by a user with the messages, set numbers and message numbers using a favorite editor. This file is then converted into the message catalog using commands described later. When changes are to be made to the message catalog, they are made to the message text source file.

The best way to describe the message catalog facility is by creating an example message catalog. In Figure 18-1, a sample catalog is created and used to describe the various procedures. You will also note that the example does not extensively use every possible option that is available. However, it provides an understanding of how to create a usable message facility by the user.

```
$
$ This is a Sample Message Text Source File to be
$ used as an example throughout this document
$

$quote "       This defines " to be the quote character
$
$set 1             First Set in the example
$
1 "                Message Number %d in Set 1\n         "

2 "File Not Found!"
3 "Cannot Open %s. Program %s aborting\n"

$set 3
14 "Yes, this is message number 14 in set 3. \
   Set and Message Numbers don't have to be in sequence \
   They only have to be in ascending order!!!"

15 "Final message in set 3 and this  catalog\n"
16 "Positional Parameter %3$s example %2$s --- %1$s\n"

$
$ End of example 1
$
```

Figure 18-1. Sample Message Text Source File

18.18.2 Syntax of Messages Within a Catalog

The format of a message text source file has certain definitions that must be followed. Note that the fields within the message text source file are separated by ASCII space and tab characters. Any other ASCII spaces or tabs are considered as being part of the subsequent field.

A line beginning with a '$' followed by ASCII space or tab characters defines a comment line. Comment lines are not included in the message catalog so any characters are allowed on a comment line. You will also notice lines that are completely blank. You can think of these lines as comment lines with nothing on them; they are not included in the message catalog either.

Throughout the example, you will notice text on some lines that doesn't look as though it is a part of the line. Any delimiters on a line (delimiters are separated by spaces and tabs remember) that follow the required syntax of the command are treated as comments. Using the line beginning with "$set 1" as an example, the characters on the line "First Set in the example" are considered comments because the syntax of this line specifies only having '$set <n>'. The delimiter in this case is the spaces/tabs after the '1'.

Following the first set of comment lines is a line that contains '$quote "'. This defines a quote character for the message text source file which can be used to surround message text so that trailing spaces or empty messages are visible in a message source line. A good example of the quote character being used is in message 1 located 2 lines below the line that starts with '$set 1'. Notice that with the quote character defined, we can put leading and trailing spaces in the message. If we had not defined a quote character, then the leading and trailing characters would have not been included in this message when the message catalog was created.

A message text source file can contain many sets to segregate the different types of messages that you will have. For example, you may want to have a set that contains your low level system messages, like "file not found", "can't execute the application specified", and have another set that contains more descriptive help messages that the user will interact with; "please hit the <foo> key to receive more help" and "to change the color of the screen, please go to the previous menu and select the color desired" are some other examples.

To create different sets, the **$set <n>** command is used. This line specifies the set identifier of the following messages until the next **$set <n>**, **$delset <n>** or end of file appears. The **<n>** parameter specifies the set identifier. Set identifiers must be presented in ascending order within a message text source file, but do not need to be contiguous. In our example, we start with set 1, then define set 3. We don't have a set 2, and there is no requirement to have it. However, it is important to note that we could set 3 first, before set 1. Messages with sets are defined as **<m> <message text>**, where **<m>**specifies the message number within the set and **<message text>** specifies the message text. The same rules apply for message numbers as applied for sets. The messages must be presented in ascending order with a set but do not need to be contiguous. In set 3 of our example, we start with message 14, followed by message 15. The messages are in ascending order, but they do not start with 1. Note that these requirements are for within sets only. Starting a new set is like starting at the beginning of the message text source file. In our example, we define set 1 to have 3 messages, 1, 2, and 3. In set 3 we start with message 14, but we could have easily started with message 1.

When defining a message, if the message text is empty and there are ASCII spaces or tabs defining a field separator, an empty string is stored in the message catalog for that message number. If there are no ASCII spaces or tabs that follow the message number, then that message is deleted from the message catalog. Finally, there is a way to continue a message on more than one line. In our example, message 14 within set 3 is on 3 lines. To continue a message on another line, a backslash (\) character is specified as the last character on the current line. Be careful when trying to continue lines though; any character following a backslash means to treat the character literally. Specifically, if a space follows the backslash, the space character is treated as a literal space (which is still a space). The message is not continued on the next line and an attempt will be made to interpret the text on the next line in the context of the message catalog. An error may occur, or you may have defined a new set or message if you included the right syntax.

18.18.3 Arguments within Messages

The actual content of the messages within a message text source file can contain text and some formatting strings. The text is obvious, the formatting options need to be defined more explicitly.

Text strings can contain special characters and escape sequences defined as follows:

newline	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
form-feed	\f
backslash	\\
octal digit	\ddd
hexidecimal digit	\xdddd

Most of these special characters have obvious meanings. They perform the same actions as in the C programming language. The specification for a hexadecimal digit allows you to define 1 or 2 hexadecimal digits . This is an effective way to get characters within your message text that you cannot enter with a keyboard. Being able to specify 2 hexadecimal digits gives you the ability to create multibyte characters from the binary counterparts.

All of the conversion characters as defined for the **printf** and **scanf** routines are also available in messages since the message text will almost always be output from a C program . An added feature called positional parameters can also be included with the conversion characters. Positional parameters allow you to specify which argument is to be used with the associated conversion character.

An example of the use of a positional parameter is message number 16 in set 3 of our example. Notice the added syntax when defining the conversion characters for printing a character string. In this message, %3$s says to use the third argument passed along with this format string and insert it at this point as a string of characters. The %2$s and %1$s perform the same action for the second and first arguments.

The advantage of using positional parameters becomes clear when a message catalog is translated into another language that use a different syntax. The position of arguments in a string that is printed in English may have to be switched when the same string is translated into Japanese. Positional parameters enable the actual arguments to stay in the same order; all that needs to change is the positional parameter in the format string when translating the message.

If you have two different conversion characters in a message string and you use positional parameters undefined results may occur. For example if a message text is of the form:

```
  "This %2$d is a test %1$s\n"
```

and the arguments passed are 10 and "foobar", then this will work. But if the positional parameters are switched to be:

```
  "This %1$d is a test %2$s\n"
```

and the arguments are the same, the string argument will be printed as a decimal value (which means the address of the beginning of the string will be printed) and the numeric value will be printed as a string.

18.18.4 Using Symbolic Definitions

Creating a message text source file with set numbers and message numbers can be a confusing process since it deals with numeric values to define messages. The message facility under AIX has a mechanism that allows the user to define symbolic values to the sets and message numbers in the message text source file. Using the **mkcatdefs** program, the symbolic values are replaced with numeric values. An include file is also created with the symbolic to numeric value relationship so the user can use the symbolic values also in the C program. For example, the following message text source file is created with symbolic values. It is exactly the same as example 1, but symbolic values instead of numeric values are used. Afterwards, an example of the header file produced is shown. The result of running **mkcatdefs** to produce this include file is the message text source file in example 1.

```
$
$ This is a Sample Message Text Source File to be
$ used as an example throughout this document
$

$quote "      This defines " to be the quote character
$
$set SET_ONE   First Set in the example
$
M_ONE_ONE   "          Message Number %d in Set 1\n    "

M_ONE_TWO   "File Not Found!"
M_ONE_THREE "Cannot Open %s. Program %s aborting at line %d\n"

$set SET_THREE
M_THREE_FOURT "Yes, this is message number 14 in set 3. \
   Set and Message Numbers don't have to be in sequence \
   They only have to be in ascending order!!!"

M_THREE_FIFT "Final message in set 3 and this catalog\n"
M_THREE_SIXT "PositionalParameter %3$s example %2$s-%1$s\n"

$
$ End of example 1
```

Figure 18-2. Message Text Source File with Symbolic Values

```
#include <limits.h>
#include <nl_types.h>
#define MF_MSGFAC "example_msg.cat"

/* The following was generated from the example symbolic message catalog */

/* definitions for set SET_ONE */
#define SET_ONE 1

#define M_ONE_ONE 1
#define M_ONE_TWO 2
```

```
#define M_ONE_THREE 3

/* definitions for set SET_THREE */
#define SET_THREE 2

#define M_THREE_FOURT 1
#define M_THREE_FIFT 2
#define M_THREE_SIXT 3
```

Figure 18-3. Include File for Symbolic Message Catalog

Notice that the numeric values are arbitrary. By including this header file in a program that will access the message catalog being created, symbolic values can be used to reference set and message numbers. The symbolic values can be arbitrary and it is recommended to make them similar to the type of message being output. For example, using **M_FILE_NOT_FOUND** would be a reasonable value for the message "file not found\n".

18.18.5 Default Values and Limits

There are various limits for the size of the message text, number of sets and number of messages that can be used in a message text source file. To determine these limits, consult the include file **<limits.h>** and **<nl_types.h>**.

18.18.6 Generation of a Message Catalog from a Message Text Source File

The syntax and acceptable values that go into making a message text source file must be converted into a message catalog. To complete this task, AIX provides the following commands:

gencat

runcat

mkcatdefs

18.18.7 gencat, runcat, mkcatdefs

The binary image of the message catalog is created by the **gencat** command. The **gencat** routine reads text and commands which describe sets and messages within sets from a text file and creates the binary image of the catalog. The **gencat** command takes as arguments a message catalog name, and a series of optional message catalog descriptor files. If our example 1 message text source file is named example.1 and we would like to create a catalog called **example.cat**, the appropriate command would look like:

```
gencat example.cat example.1
```

AIX provides the **mkcatdefs** command to convert this file into the message text source with numerical values and to create the header file with the symbol to numeric definitions. The **mkcatdefs** utility preprocesses a descriptor file that contains ASCII symbolic message identifiers (set and message numbers), producing a descriptor file that has these symbols replaced by numeric values. A header file is also produced which can be included within a user program to reference the messages symbolically. The syntax for **mkcatdefs** is to specify the prefix of the header file to be created, the suffix always being _msg.h, and the descriptor file which is a message text source file with symbolic values. The output from **mkcatdefs** is a header file called <**cmd_name**>_msg.h, the modified descriptor file with numerical values replacing the symbolic values being written to standard output. In the example, if our symbolic message text source file is called **symbol.msg**, and we want to create a header file known as **symbol_msg.h**, the the command line would look like:

```
mkcatdefs symbol symbol.msg
```

The modified descriptor file is written to standard output. We need to save this output in a file or pipe it directly to the **gencat** command to create the actual message catalog. To perform all these steps, a command would look the following:

```
mkcatdefs symbol symbol.msg | gencat example.cat
```

This would create a header file known as **symbol_msg.h** and a message catalog known as **example.cat** from the original symbol message text source file **symbol.msg**. AIX also includes a shell script known as **runcat** which converts a symbolic message text source file into a message catalog and creates the appropriate header file. The appropriate command to implement **runcat** is:

```
runcat example symbol.msg
```

18.18.8 Displaying Messages from Message Catalogs

Subtopics
18.18.8.1 dspcat, dspmsg

18.18.8.1 dspcat, dspmsg

Once the message catalog is created, you must verify that it was created correctly. To do this, use the **dspcat** command. **dspcat** displays messages in a catalog. A specific message within a specific set can be shown, or the entire catalog can be shown. The basic syntax of **dspcat** is to specify the message catalog file, along with the set and message number to be displayed. If no set or message number is given, the entire catalog is displayed.

Another very useful command is the **dspmsg** command. This command displays a message from a message catalog, including a default message if the specified message cannot be accessed. It replaces the **echo** command in shell scripts. **dspmsg** can access messages from a catalog and substitute string arguments into the message as specified on the command line. In addition to the default message, conversion arguments can be in the default message, so the arguments to the **dspmsg** command are substituted into the default message. However, one restriction is that the arguments can only be string arguments.

As an example, suppose we wanted to print out message 3 in set 1 in example 1. The command:

```
dspmsg -s 1 example.msg 3 'default %s message %s' "hello" "world"
```

would print out the following:

```
Cannot Open hello. Program world aborting at line
```

If the following bad set was specified:

```
dspmsg -s 99 example.msg 3 'default %s message %s.' "hello" "world"
```

the following message would be received:

```
default hello message world.
```

18.18.9 Accessing Messages from Message Catalogs from User Programs

The following is an example of how to access message catalogs using library routines within a C program. The following example will be used to demonstrate the appropriate library routines.

```
#include <stdio.h>
#include <locale.h>
#include <nl_types.h>
#include <limits.h>
#include "symbol_msg.h"

nl_catd catd;
extern char *malloc();

main(argc, argv)
int argc;
char **argv;
{
      char *buf, *cp;
      char mybuf[NL_TEXTMAX];

      setlocale(LC_ALL, "");
     /*
     * sequence 1:
     *     open the catalog, access a message from the catalog
     *     using different library routines. close the catalog.
     */
     if ((catd = catopen("example.cat")) == CATD_ERR) {
           printf("Can't open message catalog\n");
           exit(1);
     }

     printf(catgets(catd, SET_ONE, M_ONE_ONE, "default message"));
     printf(catgetmsg(catd, SET_ONE, M_ONE_ONE, &mybuf, NL_TEXTMAX));
     catclose(catd);

    /*
    * sequence 2:
    *    open the catalog using NLcatopen and access a
    *    message using NLcatgets(). Close the catalog
    */
    catd = NLcatopen("example.cat");
    printf(NLcatgets(catd, SET_ONE, M_ONE_ONE, "default message"), M_ONE_ONE
    catclose(catd);

  /*
  *  sequence 3:
  *    read a message from the catalog using NLgetamsg().
  *    this opens, reads the message, and closes the catalog.
  */
      printf(NLgetamsg("example.cat", SET_ONE, M_ONE_ONE, "default message")
}
```

Figure 18-4. Sample C Source Code

18.19 Use of Regular Expressions with International Characters

This section describes how to use regular expressions with international characters. The following library routines are discussed:

libPW

- **regex**
- **regcmp**

NLregexp.h

- **compile**
- **step**
- **advance**

Regular expression syntax varies slightly between the **NLregexp** and the **libPW** routines. The syntax for the **libPW regex** regular expressions is as follows:

[ ]* . ^ 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 "\" to match a new-line character.

+ (**regex** only) 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) and "*" (asterisk) operations are equivalent to "{1,}" and "{0,}", respectively.

Note: The **NLregexp.h** syntax is \{m\} \{m, \} \{m,u\}

(...)$n This stores the value matched by the enclosed regular expression in the (n+1)(th) ret parameter (**regex** only). Ten enclosed regular expressions are allowed. **regex** makes the assignments unconditionally.

Note: The **NLregexp.h** syntax is \(. . .\). The char pointers braslist[i] and braelist[i] mark the beginning and end (respectively) of the text

matching the pattern enclosed by the i'th pair of parentheses. \i matches the text matched by the i'th pair of parentheses. Paren numbers are assigned starting with the first open paren being number 1, the second open paren being number 2, etc. Parentheses nesting is allowed.

(...) (**regex** only) 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 "\" (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 (i.e. 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 'c' and 'd'. The regular expression '[c[.ch.]d]amp' would match the words camp, champ, and damp.

The **ctype** sequences ([:alpha:], etc.) can not be used as end points of a range.

The **NLregexp.h** functions compile, step and advance operate on file code strings. The following macros must be defined prior to including NLregexp.h:

Note: The examples given are from grep.c

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()

```
(PEEKC(),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 return. The **val** parameter is an error number. The error values and their meanings are:

```
#define ERROR(c)
          regerr(c)
```

Figure 18-5. Error Values

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	Weird multibyte char.

18.20 Using The C Language MBCS Interface

This section contains definitions and examples of how to interface to the MBCS library routines in **libc.a**.

Byte
An eight bit data object. Exactly the same as the C data type '**char**'.

Character
A single display object. An ASCII character is represented by exactly 7 bits. A PC 850 character is represented by exactly 8 bits. A SJIS character is represented by 8 or 16 bits, depending on the particular character. Some character sets use exactly 32 bits to represent a single character.

Character Set
A particular ordering of the characters necessary for a language or locale. Usually includes printable characters (glyphs) and non-printing (control) characters. Each character in a character set is assigned a unique (ordinal) value, and can be assigned a weight in one or more collating sequences.

File Code Character
A single display object. The representation of a character depends upon the character set in use. A file code character does not correspond directly to any C data type, but can be represented by a char (for 7 or 8 bit character sets) or as an array of chars (for character sets that require more than 8 bits). For convenience, the character type '**mbchar_t**' can be used to hold all the bytes that make up a file code character, but the representation for a particular glyph may vary depending on the file code in use.

Process Code Character
A single display object. Each process code character is represented by exactly 32 bits. There are enough process code characters to unambiguously map characters from all known languages into unique numeric values. Process Code can be thought of as a character set which can possibly encompass all other character sets. Process code characters are exactly equivalent to the C data type '**wchar_t**'.

Subtopics
18.20.1 Using The Library Routines

18.20.1 Using The Library Routines

Subtopics
18.20.1.1 Naming Conventions

18.20.1.1 Naming Conventions

The following naming conventions apply to all MBCS library routines:

Figure 18-6. Naming Conventions

Routine Name	**Deals With**	**Usually Takes**
c	bytes	char
wc	process code	wchar_t
mb	file code	mbchar_t
str	byte strings	char *
wcs	wchar_t string	wchar_t *
mbs	file code strings	char *

18.21 Getting Wide Characters (wchar_t's)

Subtopics
18.21.1 Single Character Input
18.21.2 Character Strings
18.21.3 Formatted Input

18.21.1 Single Character Input

getwchar() and **fgetwc()** parallel the stdio routines **getchar()** and **fgetc()**. To process all characters from stdin as wide chars:

```
#include <stdio.h>      /* for getwchar & WEOF */
#include <locale.h>     /* for LC_ALL */
main()
{
     wchar_t wc;
     void process_wchar(wchar_t);
     setlocale(LC_ALL,");        /* adopt current locale */
     while (WEOF != (wc = getwchar()))
     {
            process_wchar(wc);
     }
}
```

18.21.2 Character Strings

getws() and **fgetws()** parallel the stdio routines **gets()** and **fgetws()**. To process a file as lines of text:

```
#include <stdio.h>
#include <locale.h>
main()
{
     wchar_t text[256]; /* space for 256 characters */
     FILE * fp;
     void process_wstr(wchar_t *);
     setlocale(LC_ALL,"");
     fp = fopen("data_file","r");
     while (NULL != fgetws(text, sizeof(text)/sizeof(text[0]), fp))
     {
            process_wstr(text);
     }
}
```

As in **fgets**, the string is terminated by a null character. A null **wchar_t** is equivalent to L'\0', (**wchar_t**)0, or usually just plain 0. Notice the use of 'sizeof()' to determine the count parameter for **fgetws()**. **fgetws()** requires the count parameter to be in units of **wchar_t**'s. sizeof(text) returns the size of the entire array 'text' in bytes (not **wchar_t's**). We divide the size in bytes by the size of the first element of 'text' (text[0], a **wchar_t**) to obtain the number of elements (**wchar_t**'s) in the array.

18.21.3 Formatted Input

There are two new flags allowed in the **scanf()** format string, %ws and %wc:

%ws Takes a wide character string as its target, and using the current locale it converts the input to process code placing it in the target.

%wc Takes a single wide character as its target.

To accept a phone list (names and phone numbers) from stdin:

```
#include <stdio.h>
#include <locale.h>
#define MAXNAME 256              /* widest name allowed */
main()
{
     wchar_t name[MAXNAME];
     long phone_number;
     void process_pair(wchar_t *, line);
     setlocale(LC_ALL,"");
     while (EOF != scanf("%ws %ld , name, &phone_number))
     {
            process_pair(name, phone_number);
     }
}
```

18.22 Processing Wide Characters

Subtopics

18.22.1 ctype.h

All the standard **ctype.h is...()** functions are available for wide characters. Their names are similar to the **ctype** macros, except that the predicate is preceded by the letter 'w'. For example, **isalpha()** for wide characters is spelled **iswalpha()**. Likewise for all the **is...()** predicates. The **isw...()** predicates take a **wchar_t** as their parameter. The traditional **ctype is...()** predicates take ASCII characters only. DO NOT pass **wchar_t**'s to **ctype is...()** macros. To determine the **ctype** attributes of a stream of characters from stdin:

```
#include <stdio.h>
#include <ctype.h>
#include <locale.h>
main()
{
     wchar_t wc;
     while (WEOF != (wc = wgetchar()))
     {
            printf("Value: %ld =",wc);
            if (iswalpha(wc))
                 printf(" alpha");
            if (iswdigit(wc))
                 printf(" digit");
            if (iswxdigit(wc))
                 printf(" xdigit");
            if (iswpunct(wc))
                 printf(" punct");
            /*
            **   et cetera
            */
            if (iswprint(wc))
                 printf(" print );
         }
}
```

18.22.2 The String Library

All the standard string functions (as defined in **string.h**) are provided for **wchar_t** strings. The functions names contain 'wcs' where the original name contained 'str'. For example, '**strstr()**' for wide strings is named '**wcstok()**'. Routines which traditionally expected a char parameter now expect a **wchar_t**. For example, '**strchr()**' expects a **char *** and a **char**; however, '**wcschr()**' expects a **wchar_t *** and a **wchar_t**.
To print the lines from stdin matching a certain pattern, (a simplified **fgrep** command):

```
#include <stdio.h>
#include <locale.h>
#include string.h>
#define MAXLINE 256
main()
{
     wchar_t * pattern[] = "Something to look for";%
     wchar_t line[MAXLINE];
     setlocale(LC_ALL,"");
     while (NULL != getws(line))
     (%
            if (wcswcs(line,pattern))
                 putws(line);
     }
}
```

18.22.3 Conversion Routines

Sometimes it is desirable to convert from process code to file code, and vice versa. There are two main routines provided for this purpose: **wcstombs()** and **mbstowcs()**. As is implied by their names, **wcstombs** converts wide character (process code) strings to multibyte character (file code) strings, and **mbstowcs** converts from multibyte character strings to wide character strings.

Variants exist to convert from a wide character to a multibyte character string (**wctomb()**), convert from a multibyte character string to a wide character (**mbtowc()**). The following example performs the equivalent of **getws()**:

```
#include <stdio.h>
#include <stdlib.h>              /* get prototype for mbstowcs */
wchar_t *
getwstring(wchar_t *buffer)
{
     char tmp[MAXSIZE];
     if (NULL == fgets(tmp,MAXSIZE,stdin))
            return (wchar_t *) NULL;        /* fail if fgets()
                                               fails */
     if (-1 == mbstowcs(buffer, tmp, MAXSIZE))
            return (wchar_t *) NULL;        /* fail if file
                                               code is not
                                               recognized */
     return buffer;
}
```

18.22.4 Character Treatment Routines

Since process code characters are fixed size types, it is possible to access strings of process code characters using traditional pointer dereferencing operators. File code strings, however, are composite types. Accessing a string of file code characters requires functions which understand the length (in bytes) of a given character. The most useful of these functions are **mblen()** and **mbsadvance()**.
The following counts the number of characters in a multibyte character string (this is very similar to **mbslen()**):

```
int
mb_string_length(char * mbs)
{
     int len=0;
     while (*mbs)
     {
            mbs=mbsadvance(mbs);  /* mbsadvance returns
                                     NULL if string
                                     contains an invalid
                                     character */
            if (!mbs)
                 return -1;       /* fail if invalid
                                     string */
            len++;
     }
     return len;
}
```

18.22.5 Printing Wide Characters

Wide characters (**wchar_t**'s) are generally never written to a file. I/O routines are provided for converting process code into file code in such a way that programs can be written with no knowledge of the prevailing file code set whatsoever.

18.22.6 Single Character Output

putwchar() and **fputwc()** parallel the stdio routines **putchar()** and **fputc()**. To copy stdin to stdout (sort of an inefficient **cat** command):

```
#include <stdio.h>      /* for getwchar, putwchar, WEOF */
#include <locale.h>     /* for LC_ALL */
main()
{
     wchar_t wc;
     setlocale(LC_ALL,");        /* adopt current locale */
     while (WEOF != (wc = getwchar()))
     {
            putwchar(wc);
     }
}
```

18.22.7 Character Strings

putws() and **fputws()** parallel the stdio routines **puts()** and **fputws()**. To process a file as lines of text, then print the resulting lines to stdout:

```
#include <stdio.h>
#include <locale.h>
main()
{
     wchar_t text[256&rbrk; /* space for 256 characters */
     FILE * fp;
     void process_wstr(wchar_t *);
     setlocale(LC_ALL,"");
     fp = fopen("data_file","r");
     while (NULL != fgetws(text, sizeof(text)/sizeof(text[0]), fp))
     {
            process_wstr(text);
            fputws(text,stdout);
     }
}
```

Notice that we use **fputws()** to write to stdout. **Putws()** adds a newline to the end of the string when it writes to stdout, in cooperation with **getws()** which strips the newline at the end of the string. Since we used **fgetws()** to read the string, we must use **fputws()** to avoid adding an extra newline to the string.

18.22.8 Formatted Output

There are two new flags allowed in the **printf()** format string, %ws and %wc.

%ws Takes a wide character string as its input, and using the current locale it converts the input to file code and print the result. The meaning of width is modified by including the '#' flag in the format specifier. In presence of the '#' flag the width stands for the number of character otherwise it stands for the number of bytes.

%wc Takes a single wide character as its argument.

To print the minimum, maximum, and average a list of (name, value) pairs:

```
#include <stdio.h>
#include <locale.h>
#define MAXNAME 256               /* widest name allowed */
main()
{
     wchar_t name[MAXNAME];
     wchar_t name_of_max[MAXNAME] = "Error";
     wchar_t name_of_min[MAXNAME] = "Error";
     double value;
     double max = -99999.0;
     double min = 99999.0;
     double total=0.0;
     int count=0;
     setlocale(LC_ALL,"");
     while (EOF != scanf("%ws %g", name, &value))
     {
            count++;
            total += value;
            if (value > max)
            {
                 max = value;
                 wcscpy(name_of_max,name);
            }
            if (value < min)
            {
                 min = value;
                 wcscpy(name_of_min,name);
            }
     }
     printf("Maximum: %ws = %g", name_of_max);
     printf("Minimum: %ws = %g", name_of_min);
     printf("Average over %d samples: %g",count, total/count);
}
```

19.0 Chapter 19. Using Remote Procedure Call (RPC)

Subtopics

19.1 CONTENTS

19.2 About This Chapter

This chapter describes how to write network applications using the Remote Procedure Call (RPC) specification. RPC is the basic communication protocol used by the Network File Service (NFS). Remote procedure calls are used to communicate with machines in local area network usually outside your TCF cluster. This allows users to share files or access additional disk space without leaving their work stations. To access remote files within your TCF cluster does not require RPC. For additional information, refer to the discussion on Transparent Computing Facility in *Using the AIX Operating System*.

The chapter begins with an overview of the RPC specification, and the software components that support it: Remote Procedure Call Language (RPCL)and eXternal Data Representation (XDR). Next in the chapter is a discussion of the three levels of the RPC interface and how to authenticate the remote procedure calls. The chapter concludes with sections that contain examples of specific network applications of the RPC interface.

For information on specific RPC routines, see the Remote Procedure Call and RPC Service Routines sections in *AIX Operating System Technical Reference*.

19.3 Overview of RPC

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.

Subtopics

19.3.1 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.

The 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 (or blocks) for 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 awaits 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.

19.3.2 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.

19.3.3 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.

19.3.4 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.

19.3.5 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 19-1:

Figure 19-1. 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**.

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**.

19.3.6 Remote Procedure Call Language (RPCL)

RPC uses the Remote Procedure Call Language (RPCL) as the input language to its protocols and routines. RPCL specifies the data types used by RPC and generates the XDR routines that standardize their representation.

In order to implement the service protocols and routines, the RPCL input is compiled into the corresponding C language code using the **rpcgen** command. The C code is compiled by converting the RPCL definitions to C language definitions and placing them in a header file. The **rpcgen** command also compiles the corresponding XDR routine that serializes the protocol. The RPCL input can contain comments and preprocessor directives, but **rpcgen** ignores comments in the text and copies the directives to the output header file without interpreting them. For more information on the **rpcgen** command, see *AIX Operating System Commands Reference*.

The following section contains brief definitions of the RPCL syntax intended to help you understand RPCL. These definitions are not exact statements of the language protocols.

Primitive Data Types: RPCL uses the following primitive data types:

char A single character value. A **char** value can accept a sign (positive or negative).

unsigned_char A single character value on which a sign extension cannot occur.

int An integer or numerical representation.

unsigned_int An integer value that is not negative and has the same number of bits as an **int**.

long An integer value that has the same or a larger number of bits as an **int**.

unsigned_long An integer value that is not negative and has the same or a larger number of bits as an **unsigned_int**.

short An integer value that has the same or a smaller number of bits as an **int**.

unsigned_short An integer value that is not negative and has the same or a smaller number of bits as an **unsigned_int**.

float Single floating-point number value. A single floating-point number is a number that contains an exponent value. It can also contain a fraction.

double A floating-point number value that has the same or a larger number of bits as a **float**.

void No data structure. The **void** declarations can appear only inside **union** and **program** definitions.

bool Boolean value. The **rpcgen** command converts **bool** declarations to **int** declarations in an output header file.

Note: The prefix **unsigned** can be shortened to **u_** so the data types can be

abbreviated as follows: **u_char**, **u_int**, **u_long**, and **u_short**.

Defining Other Data Types: When you use RPCL data declarations, you cannot declare multidimensional arrays. You cannot point to pointers inline, but you can declare them with **typedef** statements. In addition, you can declare opaque data and strings as vectors. The vector declaration for opaque data is as follows:

```
opaque objectident +
[size]
```

A vector resulting from an opaque data declaration is compiled in the output header file as a character array of the size, in bytes, specified by the **size** parameter. Do not confuse this array declaration with the declaration of character sizes of XDR characters because the XDR characters are defined as 32-bits. The vector declaration for string data is as follows:

```
string objectident   [ maxsize ]
```

The **maxsize** parameter specifies the maximum size, in bytes, of the vector representing the string. If you do not specify **maxsize,** there is no limit to the maximum length of the string. String declarations are compiled into a character string pointer that points to the object. The character string appears in the following form:

```
string *object-ident
```

Structure and Type Definitions: The only way to generate an XDR routine is to define a data type. For example, if you define a data type named **zetype**, an XDR routine called **xdr_zetype** is generated to serialize it. You cannot nest data type definitions. Nesting them can cause the **rpcgen** command to fail when it tries to compile the C code for the definition.

Declaring Arbitrary Types with typedef: The **typedef** declarations are similar to the **typedefs** in the C language. They take the following simple declaration form:

```
typedef declaration ;
```

The **declaration** parameter contains the **typename** and **objectident** parts from the simple declaration form. The **typename** parameter specifies the name of the data type from which the new data type is derived. The **objectident** parameter specifies the name of the new data type.

Declaring Enumerations: The **enumeration-def** declarations are similar to their C language counterparts. They take the following form:

```
enum enumident {
     enumlist
};

enumlist:
     enumsymbol-ident [ = assignment ]
     enumsymbol-ident [ = assignment ], enumlist
```

The **assignment** parameter is an integer or a symbolic constant. If there is no explicit **assignment**, the implicit **assignment** is the value of the previous enumeration plus 1. If not explicitly assigned, the first

enumeration receives the value 0.

Declaring Structures: The **structure-def** declarations are similar to their C language counterparts. They take the following form:

```
  struct struct-ident   {
       declaration-list
  };

  declaration-list:
       declaration   ;
       declaration   ; declaration-list
```

Declaring Variable Length Arrays: The **variable-length-array-def** declarations are unique to RPCL. They take the following form:

```
  array array-ident {
       unsigned length-identifer ;
       vector-declaration ;
  };
```

A variable length array is similar to a structure. The following example shows a variable length array definition:

```
  array mp_int {
       unsigned len;
       short val[MAX_MP_LENGTH];
  };
```

compiled into the following structure definition:

```
  struct mp_int {
       unsigned len;
       short *val;
  };
  typedef struct mp_int mp_int;
```

Declaring Discriminated Unions: The **discriminated-union-def** declarations differ from the standard C union. The discriminated unions are unions that have a value that is added as an arm of the union to specify a particular object. RPCL discriminated union definitions take the following form:

```
  union union-ident switch (discriminant-declaration) {
  case-list
  [ default : declaration ; ]
  };

  case-list:
       case case-ident : declaration ;
       case case-ident : declaration ; case-list
  discriminant-declaration:
       declaration
```

The **union** definition appears as a cross between a C union and a C switch. The following is an example of a union definition:

```
  union net_object switch (net_kind kind) {
```

```
  case MACHINE:
       struct sockaddr_in sin;
  case USER:
       int uid;
  default:
       string whatisit;
  };
```

---

compiled into the following:

```
  struct net_object {
       net_kind kind;
       union {
            struct sockaddr_in sin;
            int uid;
            char *whatisit;
       } net_object;
  {;
  typedef struct net_object net_object;
```

Note that the output structure's **union** component name is the same as the data type name.

Declaring Program Definitions: The **program-def** declarations do not define data type. They define information used by the client programs to reference remote procedures. The **program-def**s take the following form:

```
  program program-ident {
       version-list
  } = program-number ;

  version-list:
       version
       version version-list

  version:
       version version-ident {
            procedure-list
       } = version-number ;

  procedure-list:
       procedure-declaration
       procedure-declaration procedure-list

  procedure-declaration:
       type-name procedure-ident +
       (type-name) = procedure-number;
```

Client programs use program definitions from these declarations to reference the remote procedures associated with a user commands execution. When a **program-def** is compiled, it becomes a **#define** statement that identifies a remote procedure. When the server receives the local

procedure, it matches the function of the procedure to an existing C function of the same name. Although the C function has the same name, it appears in lowercase letters and can be followed by a version number.

Note: XDR recursively frees the argument after getting the results from your local procedure, so be sure to copy from the argument any data you need between calls.

The following example illustrates the use of the **program-def** declarations. Suppose you wanted to create a server that can get or set the date. You can use **program-def** declarations to identify the remote procedures the program needs to access. The **program-def** declaration for this example could appear as follows:

```
program DATE_PROG {
     version DATE_VERS {
          date DATE_GET(timezone) = 1;
          void DATE_SET(date) = 2;
     } = 1;
} = 100;
```

This compiles in the header file as the following:

```
#define DATE_PROG 100
#define DATE_VERS 1
#define DATE_GET 1
#define DATE_SET 2
```

In the example, the local procedure defined by **#define DATE_GET** has the following form:

```
date *
date_get_1(tz)
     timezone *tz;
{
     static date d;

     return(&d);
}
```

19.3.7 Defining Arbitrary Data Types with eXternal Data Representation (XDR)

RPC handles arbitrary data structures in the remote procedure call message data by converting the structures to a network standard called eXternal Data Representation (XDR) before sending them through the system. The process of converting data from a machine's representation to the XDR format is called ***serializing***. As data is serialized, the size of each type is set to allow the data to be shared over the network independent of machine type or structure alignment algorithm. XDR serializing defines data externally in memory, allowing other functions using the same data to access the definitions.

The reverse process of serializing is called ***deserializing***. This converts the message data back to the original machine representation. XDR converts data quantities in multiples of 4 bytes when deserializing.

XDR routines are not direction-dependent. The same routines can be called to serialize and deserialize data types.

XDR routines return a non-zero number (or the value for **true** in the C language) upon successful completion. They return zero if the serialization or deserialization is not successful.

Data type parameters in RPC calls can be supplied from XDR's built-in routines, or from routines that you create. XDR contains the following set of built-in routines for the primitive data types:

xdr_int()
xdr_u_int()
xdr_long()
xdr_u_long()
xdr_short()
xdr_u_short()
xdr_bool()
xdr_enum()
xdr_string()

Note: For specific information about these routines, see *AIX Operating System Technical Reference*.

Users can create their own XDR routine to serialize a data type that they have defined. For example, you can define a data type called **simple** with a structure declaration as shown in the following:

```
------------------------------------------------------------------------

  struct simple {
          int a;
          short b;
  } simple;


------------------------------------------------------------------------
```

In the **simple** structure, the parameters have integer and short integer values.

The corresponding XDR structure could be created as follows:

```
------------------------------------------------------------------------
```

```
#include <rpc/rpc.h>

xdr_simple(xdrsp, simplep)
        XDR *xdrsp;
        struct simple *simplep;
{
        if (!xdr_int(xdrsp, &simplep->a))
                return (0);
        if (!xdr_short(xdrsp, &simplep->b))
                return (0);
        return (1);
}
```

Next, you can use the **simple** structure to call a remote procedure as shown in the following:

```
callrpc (hostname, prognum, +
versnum, procnum,
     xdr_simple, &simple ...);
```

Note: Programming with the **callrpc** routine is discussed in a later section of this book. For detailed information on **callrpc** and its parameters, see *AIX Operating System Technical Reference*.

In addition to the primitive data type routines, XDR also includes the routines for the following:

xdr_array() Arrays of arbitrary elements of fixed length.

xdr_bytes() Variable arrays of character bytes.

xdr_reference() Pointers to other structures within structures.

xdr_union() Discriminated unions which are comprised of a C language union and an enumerated value that represents an arm of the union.

Note: For specific information on these routines, see *AIX Operating System Technical Reference*.

These routines cover the constructed data types, which are more complex than the primitive data types.

XDR does not provide a routine for variable length arrays. To send a variable array of integers, the following structure can be defined:

```
struct varintarr {
        int *data;
        int arrlnth;
```

```
  } arr;
```

The **xdr_varintarr()** routine is defined as follows:

```
  xdr_varintarr (xdrsp, arrp)
          XDR *xdrsp;
          struct varintarr *arrp;
  {
          xdr_array(xdrsp, &arrp->data, &arrp->arrlnth, MAXLEN,
                  sizeof(int), xdr_int);
  }
```

The corresponding XDR routine takes as parameters:

The XDR stream handle. The XDR stream handle identifies the XD stream where data objects are serialized to XDR format and deserialized back into machine representation.

Pointer to the data in the array. This is also where the dat representation is placed for future use.

Pointer to the size of the array

Maximum length of the array allowed

Size of each array element

The XDR routine that handles each array element

With the variable array of integers defined, the following remote procedure call can be made:

```
  callrpc (hostname, prognum, +
  versnum, procnum,
       xdr_varintarr, &arr ...);
```

Note: Programming with **callrpc** is discussed in a later section of this book. For detailed information on **callrpc** and its parameters, see *AIX Operating System Technical Reference*. See the XDR library section of *AIX Operating System Technical Reference* for information on XDR routines.

Subtopics
19.3.7.1 Allocating Memory With eXternal Data Representation (XDR)

19.3.7.1 Allocating Memory With eXternal Data Representation (XDR)

In addition to their input and output functions, XDR routines allocate memory. For this reason, the second parameter of XDR routines contains a pointer to the object instead of the object itself. In the case of the constructed data types, the sizes of objects are specified and placed in structures that are pointed to by the second parameter.

Allocating and freeing memory is usually not a concern unless a routine specifically directs it. However, a simplified example of memory allocation is shown below. For more information, see the XDR section in *AIX Operating System Technical Reference*.

In the following example, the **xdr_chararr1** routine deals with a fixed array of bytes with the length **SIZE**:

```
---------------------------------------------------------------------------

  xdr_chararr1(xdrsp, chararr)
           XDR *xdrsp;
           char chararr[];
  {

           char *p;
           int len;

           p = chararr;
           len = SIZE;
           return (xdr_bytes(xdrsp, &p, &len, SIZE));
  }


---------------------------------------------------------------------------
```

It can be called from a server using the following:

```
---------------------------------------------------------------------------

  char chararr[SIZE];
  svc_getargs(transp, xdr_chararr1, chararr);


---------------------------------------------------------------------------
```

where **chararr** has already allocated space.

If you want XDR to allocate the memory space, the routine can be written as follows:

```
---------------------------------------------------------------------------

  xdr_chararr2 (xdrsp, chararrp)
           XDR *xdrsp;
           char **chararrp;
  {
           int len;

           len = SIZE;
           return (xdr_bytes(xdrsp, charrarrp, &len, SIZE));
  }
```

The RPC call can look like the following:

```
  char *arrptr;
  arrptr = NULL;
  svc_getargs(transp, xdr_chararr2, &arrptr);
  svc_freeargs(transp, xdr_chararr2, &arrptr);
```

The **svc_freeargs** routine frees the character array.

For more information about the XDR routines, see the XDR library section of *AIX Operating System Technical Reference*.

19.4 Authenticating Remote Procedure Calls

RPC servers are configured to contain the information to authenticate all remote procedure calls made with RPC routines. When remote procedure calls are made, the caller sends authentication parameters with the routine as identification. ***Authentication*** is a means of verifying the user of an information system or resource. You can associate different forms, sometimes called flavors, of authentication with RPC clients just as you can set different transport protocols (UDP/IP and TCP/IP) for message delivery when you create RPC clients and servers. The default authentication parameter is the null value or can be set by the **authunix_none** routine.

Subtopics
19.4.1 Authenticating the Client
19.4.2 Authenticating the Server

19.4.1 Authenticating the Client

When a caller creates a new RPC client handle, the routine returns the authentication handle declaration as the following:

```
clnt->cl_auth = authnone_create ();
```

You can set the RPC client authentication by setting the statement after creating the RPC client handle to the appropriate routine. For example, you can set the client authentication to be the default AIX style by the following entry:

```
clnt->cl_auth = authunix_create_default();
```

This causes each RPC call associated with the client handle specified by **clnt** to carry the following authentication credentials structure:

```
/*
  * Unix style credentials.
  */
struct authunix_parms {
        u_long aup_time;          /* credentials creation time */
        char  *aup_machname;      /* host name where client is */
        int    aup_uid;           /* client's UNIX effective uid */
        int    aup_gid;           /* client's current group id */
        u_int  aup_len;           /* element length of aup_gids */
        int   *aup_gids;          /* array of groups user is in */
};
```

Figure 19-2. Example of Authentication Credentials Structure

The fields in the credentials structure are set when the **authunix_create_default** routine invokes the appropriate system calls.

The user who created the RPC authentication is responsible for destroying it to conserve memory. Use the following routine to free space used by the authentication:

```
auth_destroy(clnt->cl_auth);
```

19.4.2 Authenticating the Server

The RPC service package passes authentication parameters as opaque data. This means that the structures are passed as arbitrary types without interpretations. To understand the authentication data that is passed, examine the typical structure of an RPC request that follows:

Note: The comments to the side indicate the definition of the parameter.

```
------------------------------------------------------------------------

  struct svc_req {
          u_long rq_prog;       /* service program number */
          u_long rq_vers;       /* service protocol version number */
          u_long rq_proc;       /* desired procedure number */
          struct opaque_auth
                   rq_cred;       /* raw credentials from wire */
          caddr_t rq_clntcred;  /* credentials that specify read only */
  };


------------------------------------------------------------------------
```

The **rq_cred** parameters are mostly opaque to the programmer. The exception is the style of authentication used.

The following fields can be set by the programmer:

```
------------------------------------------------------------------------

  struct opaque_auth {
          enum_t  oa_flavor;      /* style of credentials */
          caddr_t oa_base;        /* address of more auth stuff */
          u_int   oa_length;      /* not to exceed MAX_AUTH_BYTES */
  };


------------------------------------------------------------------------
```

The service implementer can inspect the request's **rq_cred.oa_flavor** to determine which style of authentication the caller used. The request's **rq_clntcred** field is **NULL** or points to a structure that corresponds to a supported style of authentication credentials.

Consider the following regarding server authentication:

It is not necessary to check the authentication parameters associate with the **nullproc** procedure number since it means the procedure is number zero.

If the authentication parameter's type cannot be matched on you server, call the **svcerr_weakauth** routine.

The service protocol itself should return the information regardin denied access.

RPC does not use an individual service's access controls. Eac service must implement its own access control policies and reflect these policies as a return value in its protocol.

The remote users service example shown in Figure 19-3 computes the results from the remote procedure call that requests the total number of remote users minus the user identified by UID 16.

```
  #include <stdio.h>
  #include <rpc/rpc.h>
  #include <utmp.h>
  #include <rpcsvc/rusers.h>

  nuser(rqstp, transp)
       struct svc_req *rqstp;
       SVCXPRT *transp;
  {
       struct authunix_parms *unix_cred;
       int uid;
       unsigned long nusers;

       /*
        * we don't care about authentication for null proc
        */
       if (rqstp->rq_proc == NULLPROC) {
           if (!svc_sendreply(transp, xdr_void, 0)) {
               fprintf(stderr, "can't reply to RPC call\n");
               exit(1);
           }
           return;
       }
       /*
        * now get the uid
        */
       switch (rqstp->rq_cred.oa_flavor) {
       case AUTH_UNIX:
           unix_cred = (struct authunix_parms *)rqstp->rq_clntcred;
           uid = unix_cred->aup_uid;
           break;

       case AUTH_NULL:
       default:
           svcerr_weakauth(transp);
           return;
       }
       switch (rqstp->rq_proc) {
       case RUSERSPROC_NUM:
           /*
            * make sure caller is allowed to call this proc
            */
           if (uid == 16) {
               svcerr_systemerr(transp);
               return;
           }
           /*
            * code here to compute the number of users
            * and put in variable nusers
            */
           if (!svc_sendreply(transp, xdr_u_long, &nusers)){
               fprintf(stderr, "can't reply to RPC call\n");
               exit(1);
           }
```

```
            return;
        default:
            svcerr_noproc(transp);
            return;
        }
    }
```

Figure 19-3. Remote Users Service Example

19.5 Special Topics

This section discusses certain aspects related to RPC functions.

Subtopics

19.5.1 The select Procedure on the Server Side

A **select** procedure allows you to ***poll*** devices, that is, read to see if there is data present and continue regular processing if not. For example, a process can execute RPC requests while periodically interrupting regular processing to update a data structure. The process sets an alarm signal before calling the **svc_run** routine. If the process' other activity involves waiting for a file descriptor, the **svc_run** call does not work. Figure 19-4 shows the code for **svc_run**:

```
#include <errno.h>
#include <rpc/rpc.h>
void
svc_run()
{
     int readfds;

     for (;;) {
         readfds = svc_fds;
         switch (select(32, &readfds, NULL, NULL, NULL)) {

         case -1:
             if (errno == EINTR)
                 continue;
             perror("rstat: select");
             return;
         case 0:
             break;
         default:
             svc_getreq(readfds);
         }
     }
}
```

Figure 19-4. Code for the Library Routine svc_run

If you identify the file descriptors of the sockets for the programs for which you are waiting, you can bypass the **svc_run** routine and call the **svc_getreq** routine directly.

See the RPC Service Routines section in *AIX Operating System Technical Reference* for information on RPC library routines.

19.5.2 Broadcast RPC

With RPC, you can broadcast remote procedure calls across the network. When you broadcast data, it is transmitted to all servers listening at the port or ports the data is transmitted on. This means clients do not have to depend on a particular server to answer requests. Clients can also broadcast remote procedure calls in order to receive more than one answer to a service request.

Broadcast RPC is only supported by transport protocols that deliver information in packet form, such as UDP/IP.

Broadcast RPC filters out all unsuccessful responses. The user does not know if calls were discarded or not. For example, if a mismatch between the broadcast and remote service versions occurs, the responses are filtered out.

Broadcast messages are sent through the **portmap** daemon. The ***portmap*** daemon program, sometimes called the **portmapper** runs on the local servers to map RPC program program numbers to the ports that service each procedure. RPC can only access broadcast service requests that are registered with their **portmap** daemon. The **portmap** daemon, in conjunction with standard RPC protocols, is required to use broadcast RPC.

Figure 19-5 shows an example of broadcast RPC:

```
#include <rpc/pmap_clnt.h>

enum clnt_stat  clnt_stat;

clnt_stat =
clnt_broadcast(prog, vers, proc, xargs, argsp, xresults,
     resultsp, eachresult)
u_long          prog;        /* program number */
u_long          vers;        /* version number */
u_long          proc;        /* procedure number */
xdrproc_t       xargs;       /* xdr routine for args */
caddr_t         argsp;       /* pointer to args */
xdrproc_t       xresults;    /* xdr routine for results */
caddr_t         resultsp;    /* pointer to results */
bool_t (*eachresult)();      /* call with each result gotten */
```

Figure 19-5. Example of broadcast RPC

The procedure **eachresult** is called for each valid result. It returns a boolean expression that indicates whether or not the client wants more responses, as the following example shows:

```
bool_t                  done;
done =
eachresult(resultsp, raddr)
caddr_t resultsp;
struct sockaddr_in *raddr;  /* addr of responding machine */
```

If **done** is TRUE, broadcasting stops and **clnt_broadcast** returns successfully. Otherwise, the routine waits for another response. The request is rebroadcast after waiting a few seconds. If no response comes back, the routine returns with **RPC_TIMEDOUT**. To interpret **clnt_stat** errors, feed the error code to the **clnt_perrno** routine.

19.5.3 Using Batching

When you use RPC, the client sends a call message and waits for servers to reply. RPC batch facilities allow clients to continue processing while waiting for a response from a server.

When you use batching, you can place RPC messages in a pipeline of calls to a server. The pipeline of calls is carried on a reliable transport such as TCP/IP. RPC batching is more efficient than single call messages because:

- The client can continue to generate new calls while the serve executes previous ones.
- TCP/IP can send many call messages to the server in one **write** system call.

Note: No response is required from the server for each call in the pipeline.

Figure 19-7 shows an example of RPC batching with TCP/IP delivery. It has two calls that create strings. One call returns void results. The other call does not return results. Note that you must have the file **windows.h** in the directory **/usr/include/rpcsvc**. Figure 19-6 shows the contents of the file **/usr/include/rpcsvc/windows.h**.

```
Constants for the WINDOWS example RPC program
/* Program and version numbers */
#define WINDOWPROG              ((u_long)100099)
#define WINDOWVERS              ((u_long)1)

/* Procedure numbers */
#define RENDERSTRING            ((u_long)1)
#define RENDERSTRING_BATCHED    ((u_long)2)
```

Figure 19-6. Constants for the WINDOWS example RPC program

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/windows.h>

void windowdispatch();

main()
{
        SVCXPRT *transp;

        transp = svctcp_create(RPC_ANYSOCK, 0, 0);
        if (transp == NULL){
                fprintf(stderr, "can't create an RPC server\n");
                exit(1);
        }
        pmap_unset(WINDOWPROG, WINDOWVERS);
```

```
        if (!svc_register(transp, WINDOWPROG, WINDOWVERS,
                windowdispatch, IPPROTO_TCP)) {
                        fprintf(stderr, "can't register WINDOW service\n");
                        exit(1);
        }
        svc_run();  /* never returns */
        fprintf(stderr, "should never reach this point\n");
}

void
windowdispatch(rqstp, transp)
        struct svc_req *rqstp;
        SVCXPRT *transp;
{
        char *s = NULL;

        switch (rqstp->rq_proc) {
        case NULLPROC:
                if (!svc_sendreply(transp, xdr_void, 0)) {
                        fprintf(stderr, "can't reply to RPC call\n");
                        exit(1);
                }
                return;
        case RENDERSTRING:
                if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                        fprintf(stderr, "can't decode arguments\n");
                        svcerr_decode(transp);
                        break;
                }

                /*
                 * call here to render the string s
                 */
                if (!svc_sendreply(transp, xdr_void, NULL)) {
                        fprintf(stderr, "can't reply to RPC call\n");
                        exit(1);
                }
                break;

        case RENDERSTRING_BATCHED:
                if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                        fprintf(stderr, "can't decode arguments\n");
                       /*
                       * we are silent in the face of protocol errors
                       */
                       break;

                }
                /*
                 * call here to render string s, but send no reply!
                 */
                break;
        default:
                svcerr_noproc(transp);
                return;
        }
        /*
        * now free string allocated while decoding arguments
        */
        svc_freeargs(transp, xdr_wrapstring, &s);
```

```
}
```

Figure 19-7. Example of a Client Using Batching to Create Strings Using RPC Batching with TCP/IP Delivery

The server can have one procedure that takes the string and a boolean to indicate whether or not the procedure should respond.

For a client to take advantage of batching, it must perform RPC calls on a TCP-based delivery. The calls must have the following attributes:

- The result of the XDR routine must be zero (**NULL**).
- The RPC call timeout must be zero

Figure 19-8 shows an example of a client using batching to create several strings. The batch is emptied when the client gets a null string. Note that you must have the file **windows.h** in the directory **/usr/include/rpcsvc**; see page 19.5.3 for a listing of the contents of this file.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>

#include <rpcsvc/window.h>

main(argc, argv)
        int argc;
        char **argv;
{
        struct hostent *hp;
        struct timeval pertry_timeout, total_timeout;
        struct sockaddr_in server_addr;
        int addrlen, sock = RPC_ANYSOCK;
        register CLIENT *client;
        enum clnt_stat clnt_stat;
        char buf[1000], *s = buf;

        /* initial as in Figure 19-19 in topic 19.7.3.2
         */
        server_addr.sin_family = AF_INET;
        if ((client = clnttcp_create(&server_addr,
            WINDOWPROG, WINDOWVERS, &sock, 0, 0)) == NULL) {
                perror("clnttcp_create");
                exit(-1);
        }
        total_timeout.tv_sec = 0;
        total_timeout.tv_usec = 0;
        while (scanf("%s", s) != EOF) {
                clnt_stat = clnt_call(client, RENDERSTRING_BATCHED,
                        xdr_wrapstring, &s, NULL, NULL, total_timeout);
                if (clnt_stat != RPC_SUCCESS) {
                        clnt_perror(client, "batched rpc");
```

```
                        exit(-1);
                }
        }
        /* now flush the pipeline
         */
        total_timeout.tv_sec = 20;
        clnt_stat = clnt_call(client, NULLPROC, xdr_void, NULL,
                xdr_void, NULL, total_timeout);
        if (clnt_stat != RPC_SUCCESS) {
                clnt_perror(client, "rpc");
                exit(-1);
        }
        clnt_destroy(client);
}
```

Figure 19-8. Example of a Client Using Batching to Create Strings

The server does not send a message to notify clients of failures.

19.5.4 Using the inetd Daemon to Start a Server

You can start an RPC server from the **inetd** daemon. Change the original code to call the **svcudp_create** routine as follows:

```
transp = svcudp_create (0);
```

This way the **inetd** process passes a socket as a file descriptor with the value 0.

In addition, call the **svc_register** routine as:

```
svc_register (transp, PROGNUM, VERSNUM, service, 0);
```

with the final flag as the value 0, since the program is already registered by the **inetd** daemon.

Remember that if you want to exit from the server process and return control to the **inetd** daemon, you must explicitly exit because the **svc_run** routine never returns.

The format of entries in the **/etc/inetd.conf** file for RPC service procedures is as follows:

```
svc_name sunrpc_udp udp wait|nowait root +
program prognum versnum
```

The *program* field is the C code implementing the server, and *prognum* and *versnum* are the program and version numbers of the RPC procedure. The value **udp** can be replaced by **tcp** for TCP/IP based RPC services as follows:

```
svc_name sunrpc_tcp tcp nowait
```

The version number can be a range if the same program handles multiple versions, as shown in the following example:

```
rstatd sunrpc_udp udp wait root /usr/etc/rpc.rstadt +
rstatd 100001 1-2
```

19.6 Example Applications

The following sections show examples of specific network applications using RPC routines. The explanation accompanying each example briefly explains the use of each application. For a more detailed explanation of the routines and their parameters, see *AIX Operating System Technical Reference*.

Subtopics

19.6.1 Example of Using Version Numbers

An RPC version number identifies the version of an RPC program. It is possible to use several versions of the same program through a server in order to call a remote procedure. The following example shows how an RPC server can service two versions of the same program. In this example, the first version number of the program called **RUSERSPROG** is **RUSERSVERS_ORIG**. There is a new version of **RUSERPROG** that returns an unsigned short rather than an unsigned long. This version is named **RUSERSVERS_SHORT**. A server that supports both versions does a double register, as shown in Figure 19-9.

```
       if (!svc_register(transp, RUSERSPROG, RUSERSVERS_ORIG,
         nuser, IPPROTO_TCP)) {
           fprintf(stderr, "can't register RUSER service\n");
           exit(1);
       }
       if (!svc_register(transp, RUSERSPROG, RUSERSVERS_SHORT,
         nuser, IPPROTO_TCP)) {
           fprintf(stderr, "can't register RUSER service\n");
           exit(1);
       }
```

Figure 19-9. A Server Supporting Two Versions of the Same Program

A C program can be written to handle both versions. Figure 19-10 shows an example of a C program that handles both versions.

```
  #include <stdio.h>
  #include <rpc/rpc.h>
  #include <utmp.h>
  #include <rpcsvc/rusers.h>

  nuser(rqstp, transp)
       struct svc_req *rqstp;
       SVCXPRT *transp;
  {
       unsigned long nusers;
       unsigned short nusers2;

       switch (rqstp->rq_proc) {
       case NULLPROC:
           if (!svc_sendreply(transp, xdr_void, 0)) {
               fprintf(stderr, "can't reply to RPC call\n");
               exit(1);
           }
           return;
       case RUSERSPROC_NUM:
           /*
            * code here to compute the number of users
            * and put in variable nusers
            */
           nusers2 = nusers;
           if (rqstp->rq_vers != RUSERSVERS_ORIG)
```

```
            return;
        if (!svc_sendreply(transp, xdr_u_long, &nusers)) {
            fprintf(stderr, "can't reply to RPC call\n");
            exit(1);
        } else
        if (!svc_sendreply(transp, xdr_u_short, &nusers2)) {
            fprintf(stderr, "can't reply to RPC call\n");
            exit(1);
        }
        return;
    default:
        svcerr_noproc(transp);
        return;
    }
}
```

Figure 19-10. C Procedure Handling Two Versions of the Same Program

19.6.2 Example of Using Transmission Control Protocol/Internet Protocol (TCP/I

In the following example, RPC uses the Transmission Control Protocol/Internet Protocol (TCP/IP) to carry messages between communicating programs. TCP/IP transports RPC messages in long streams of data. In Figure 19-11, the initiator of the remote procedure call takes its standard input and sends it to the server. The server lists it to standard output. This example also shows an XDR procedure that behaves differently on serialization than on deserialization.

```
/*
   * The xdr routine:
   *              on decode, read from wire, write onto fp
   *              on encode, read from fp, write onto wire
   */
#include <stdio.h>
#include <rpc/rpc.h>

bool_t

xdr_rcp(xdrs, fp)
     XDR *xdrs;
     FILE *fp;
{
     unsigned long size;
     char buf[BUFSIZ], *p;

     if (xdrs->x_op == XDR_FREE)/* nothing to free */
         return 1;
     while (1) {
         if (xdrs->x_op == XDR_ENCODE) {
             if ((size = fread(buf, sizeof(char), BUFSIZ,
               fp)) == 0 && ferror(fp)) {
                 fprintf(stderr, "can't fread\n");
                 exit(1);
                                }

         }
         p = buf;
         if (!xdr_bytes(xdrs, &p, &size, BUFSIZ))
             return 0;
         if (size == 0)
             return 1;
         if (xdrs->x_op == XDR_DECODE) {

             if (fwrite(buf, sizeof(char), size,
               fp) != size) {
                 fprintf(stderr, "can't fwrite\n");
                 exit(1);
             }
         }
     }
}

/*
  * The sender routines
  */
#include <stdio.h>
```

```
#include <netdb.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>
/* program and version numbers*/
#define RCPPROG   ((u_long)100100)
#define RCPVERS   ((u_long)1)

/*Procedure nummbers*/
#define RCPPROC_FP ((u_long)1)


main(argc, argv)
     int argc;
     char **argv;
{
     int err;

     if (argc < 2) {
         fprintf(stderr, "usage: %s servername\n", argv[0]);
         exit(-1);

     }
     if ((err = callrpctcp(argv[1], RCPPROG, RCPPROC_FP,
       RCPVERS, xdr_rcp, stdin, xdr_void, 0) != 0)) {
         extern bool_t xdr_rcp ();
         clnt_perrno(err);
         fprintf(stderr, "can't make RPC call\n");
         exit(1);
     }

}
callrpctcp(host, prognum, procnum, versnum,
             inproc, in, outproc, out)
     char *host, *in, *out;
     xdrproc_t inproc, outproc;

{
     struct sockaddr_in server_addr;
     int socket = RPC_ANYSOCK;
     enum clnt_stat clnt_stat;
     struct hostent *hp;
     register CLIENT *client;
     struct timeval total_timeout;


if ((hp = gethostbyname(host)) == NULL) {
         fprintf(stderr, "can't get addr for '%s'\n", host);
         exit(-1);
     }
     bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
         hp->h_length);
     server_addr.sin_family = AF_INET;
     server_addr.sin_port =  0;
     if ((client = clnttcp_create(&server_addr, prognum,
       versnum, &socket, BUFSIZ, BUFSIZ)) == NULL) {
         perror("rpctcp_create");
         exit(-1);

     }
```

```
     total_timeout.tv_sec = 20;
     total_timeout.tv_usec = 0;
     clnt_stat = clnt_call(client, procnum,
         inproc, in, outproc, out, total_timeout);
     clnt_destroy(client);
     return (int)clnt_stat;
}

/*
  * The receiving routines
  */
#include <stdio.h>
#include <rpc/rpc.h>
/*Program and version numbers*/
#define RCPPROG ((u_long)100100)
#define RCPVERS ((u_long)1)
/*Procedure numbers*/
#define RCPPROC_FP ((u_long)1)


main()
{
     register SVCXPRT *transp;
     extern void rcp_service ();
     if ((transp = svctcp_create(RPC_ANYSOCK,
       BUFSIZ, BUFSIZ)) == NULL) {
         fprintf("svctcp_create: error\n");
         exit(1);


     }
     pmap_unset(RCPPROG, RCPVERS);
     if (!svc_register(transp,
       RCPPROG, RCPVERS, rcp_service, IPPROTO_TCP)) {
         fprintf(stderr, "svc_register: error\n");
         exit(1);

     }
     svc_run();  /* never returns */
     fprintf(stderr, "svc_run should never return\n");
}

void
rcp_service(rqstp, transp)
     register struct svc_req *rqstp;
     register SVCXPRT *transp;

{
     extern bool_t xdr_rcp ();
     switch (rqstp->rq_proc) {
     case NULLPROC:
         if (svc_sendreply(transp, xdr_void, 0) == 0) {
             fprintf(stderr, "err: rcp_service");
             exit(1);

         }
         return;
     case RCPPROC_FP:
         if (!svc_getargs(transp, xdr_rcp, stdout)) {
             svcerr_decode(transp);
```

```
            return;

        }
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "can't reply\n");
            return;
        }
        exit(0);
    default:
        svcerr_noproc(transp);
        return;
    }
}
```

Figure 19-11. Example of Using Transmission Control Protocol/Internet Protocol (TCP/IP)

19.6.3 Example of Using Callback Procedures

A server can become a client and make a remote procedure call back to a client process. An RPC callback requires a program number on which to make the RPC call. Figure 19-12 shows how the **gettransient** routine generates a program number dynamically. The **gettransient** routine returns a valid program number in the transient range (see the table in "Assigning Program Numbers to Protocols" in topic 19.3.5), and registers the number with the **portmap** daemon, sometimes called the **portmapper**. It communicates only with the **portmap** daemon running on the same machine as the **gettransient** routine. The call to **pmap_set** is a test and set operation that tests whether a program number has already been registered, and reserves the number if not.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/in.h>

gettransient(proto, vers, sockp)
     int proto, vers, *sockp;
{
     static int prognum = 0x40000000;
     int s, len, socktype;
     struct sockaddr_in addr;
     switch(proto) {
         case IPPROTO_UDP:
             socktype = SOCK_DGRAM;
             break;
         case IPPROTO_TCP:
             socktype = SOCK_STREAM;
             break;
         default:
             fprintf(stderr, "unknown protocol type\n");
             return 0;
     }
     if (*sockp == RPC_ANYSOCK) {
         if ((s = socket(AF_INET, socktype, 0)) < 0) {
             perror("socket");
             return (0);
         }
         *sockp = s;
     }
     else
         s = *sockp;
     addr.sin_addr.s_addr = 0;
     addr.sin_family = AF_INET;
     addr.sin_port = 0;
     len = sizeof(addr);
     /*
      * may be already bound, so don't check for error
      */
     bind(s, &addr, len);
     if (getsockname(s, &addr, &len)< 0) {
         perror("getsockname");
         return (0);
     }
     while (!pmap_set(prognum++, vers, proto, ntohs(addr.sin_port)))
```

```
            continue;
        return (prognum-1);
    }
```

Figure 19-12. Example of Using a Callback Procedure

The programs shown in Figure 19-13 and Figure 19-14 show how to make an RPC callback using the **gettransient** routine. The client makes an RPC call to the server, passes a transient program number, and waits to receive a callback from the server at that program number. The server registers the program **EXAMPLEPROG**, to receive the RPC call informing it of the callback program number. At some time (which in this example is when it receives **SIGALRM**), it sends a callback RPC call using the program number it received earlier.

This program is included only as an example. The user should define the specific values.

```
  /*
    * client
    */
  #include <stdio.h>
  #include <rpc/rpc.h>
  /*Example program and version numbers*/
  #define EXAMPLEPROG ((u_long)100101)
  #define EXAMPLEVERS ((u_long)1)
  /*Procedure number*/
  #define EXAMPLEPROC_CALLBACK ((u_long)1)

  int callback();
  char hostname[256];

  main(argc, argv)
      int argc;
      char **argv;
  {
       int x, ans, s;
       SVCXPRT *xprt;

       gethostname(hostname, sizeof(hostname));
       s = RPC_ANYSOCK;
       x = gettransient(IPPROTO_UDP, 1, &s);
       fprintf(stderr, "client gets prognum %d\n", x);
       if ((xprt = svcudp_create(s)) == NULL) {
         fprintf(stderr, "rpc_server: svcudp_create\n");
           exit(1);
       }
       /* protocol is 0 - gettransient() does registering
        */
       (void)svc_register(xprt, x, 1, callback, 0);
       ans = callrpc(hostname, EXAMPLEPROG, EXAMPLEVERS,
           EXAMPLEPROC_CALLBACK, xdr_int, &x, xdr_void, 0);

       if (ans != RPC_SUCCESS) {
           fprintf(stderr, "call: ");
```

```
            clnt_perrno(ans);
            fprintf(stderr, "\n");
        }
        svc_run();
        fprintf(stderr, "Error: svc_run shouldn't return\n");
    }
    callback(rqstp, transp)
        register struct svc_req *rqstp;
        register SVCXPRT *transp;
    {
        switch (rqstp->rq_proc) {
            case 0:
                if (!svc_sendreply(transp, xdr_void, 0)) {
                    fprintf(stderr, "err: callback\n");
                    exit(1);
                }
                exit(0);
            case 1:
                if (!svc_getargs(transp, xdr_void, 0)) {
                    svcerr_decode(transp);
                    exit(1);
                }
                fprintf(stderr, "client got callback\n");
                if (!svc_sendreply(transp, xdr_void, 0)) {
                    fprintf(stderr, "err: callback\n");
                    exit(1);
                }
        }
    }
```

Figure 19-13. Using the gettransient Routine with a Client

This program is included only as an example. The user should define the specific values.

```
    /*
     * server
     */
    #include <stdio.h>
    #include <rpc/rpc.h>
    #include <sys/signal.h>
    /*Example Program and version numbers*/
    #define EXAMPLEPROG ((u_long)100101)
    #define EXAMPLEVERS ((u_long)1)
    /*Procedure numbers*/
    #define EXAMPLEPROC_CALLBACK ((u_long)1)

    char *getnewprog();
    char hostname[256];
    int docallback();
    int pnum;               /* program number for callback routine */

    main(argc, argv)
        int argc;
        char **argv;
```

```
{
     gethostname(hostname, sizeof(hostname));
     registerrpc(EXAMPLEPROG, EXAMPLEVERS,
       EXAMPLEPROC_CALLBACK, getnewprog, xdr_int, xdr_void);
     fprintf(stderr, "server going into svc_run\n");
     signal(SIGALRM, docallback);
     alarm(10);
     svc_run();
     fprintf(stderr, "Error: svc_run shouldn't return\n");
}

char *
getnewprog(pnump)
     char *pnump;
{
     pnum = *(int *)pnump;
     return NULL;
}

docallback()
{
     int ans;

     ans = callrpc(hostname, pnum, 1, 1, xdr_void, 0,
         xdr_void, 0);
     if (ans != 0) {
         fprintf(stderr, "server: ");
         clnt_perrno(ans);
         fprintf(stderr, "\n");
     }
}
```

Figure 19-14. Using the gettransient Routine with a Server

19.7 RPC Programming Levels

The RPC interface is divided into the following three programming levels:

At the highest level, you can call RPC library routines directly in program. The RPC library routines take care of making the remote procedure calls they need to function.

At the middle level, you can directly use the **registerrpc** and **callrpc** routines to make and execute remote procedure calls. Use this level for most common RPC applications.

At the lowest level, you can work directly with ***sockets*** that transmit the RPC messages. In AIX, ***sockets*** are the mapping of port numbers to Internet addresses in order to create a unique identifier for a host machine. Use this level for more direct control, such as changing the defaults of RPC routines and manipulating the sockets.

Subtopics

19.7.1 Using the Highest Level

When you use the highest level of RPC, you call RPC library routines to make the procedure calls in your programs. The library routines access RPC service procedures needed to make and execute the required remote procedures associated with a remote program. RPC is transparent to the programmer at this level because it resembles programming without the need to know RPC.

When you use a library routine in a program, the program must be compiled using the **rpcsvc** library file. Compile the program with the following command:

```
cc program.c -lrpcsvc
```

The following table contains the RPC library routines available for inclusion in your programs.

Routine	Description
getrpcport()	Gets RPC port number servicing the requested procedure's function.
rnusers()	Returns number of users on remote machine.
rusers()	Returns information about users on remote machine.
havedisk()	Determines if remote machine has disk.
rstat()	Gets performance data from remote kernel.
rwall()	Writes to specified remote machines.
getmaster()	Gets name of Network Information Service (NIS) master server if NIS is being used to administer the system.
yppasswd()	Updates user password in NIS databases if NIS Pages is being used to administer the system.

Figure 19-15 shows how to use the **rnusers** library routine to determine the number of users logged in to a remote machine.

```
#include <stdio.h>

main(argc, argv)
     int argc;
     char **argv;
{
     int num;

     if (argc < 2) {
         fprintf(stderr, "usage: rnusers hostname\n");
         exit(1);
     }
     if ((num = rnusers(argv[1])) < 0) {
         fprintf(stderr, "error: rnusers\n");
         exit(-1);
```

```
        }
        printf("%d users on %s\n", num, argv[1]);
        exit(0);
  }
```

Figure 19-15. Using the rnusers Library Routine in a Program

19.7.2 Using Intermediate-Level RPC

When you use intermediate-level RPC, you can work directly with the **callrpc** and **registerrpc** routines to make and execute remote procedure calls. The **callrpc** routine calls a remote procedure. The **registerrpc** routine matches the remote procedure number to its corresponding C language procedure.

Access to a remote procedure is through the program number, version number, and procedure number. The ***program number*** defines a group of related remote procedures. Each program has a ***version number*** to allow minor changes to a program, such as adding a new related procedure, without assigning a new program number. Each of the program's related procedures has a ***procedure number*** that uniquely identifies it in the group. The program, version and procedure number related to each remote service procedure is documented by RPC. You must look up the information to call a remote service procedure.

Subtopics
19.7.2.1 Using the callrpc Routine
19.7.2.2 Using the registerrpc Routine

19.7.2.1 Using the callrpc Routine

Eight parameters are used to call a remote procedure with the **callrpc** routine. Besides the program, version and procedure numbers, the name of the remote machine on which the procedure resides must be identified. The next two parameters contain the arguments to the remote procedure call.

There are two more parameters that identify the return value of the call. One return parameter contains the return value itself. If the **callrpc** routine succeeds, it returns the value 0. If the routine does not succeed, it returns an integer value that represents a client condition. The meaning of each return code is defined in the **<rpc/clnt.h>** header file. The other return parameter points to the location of the return value.

Since internal machine data types can be represented differently on machines, you must supply the data type of the RPC argument as well as a pointer to the argument itself.

For example, the **RUSERSPROC_NUM** function takes no arguments, resides on a network server, and returns an unsigned long data type value. When you use the **callrpc** routine to call the **RUSERSPROC_NUM** function, **callrpc** uses the **xdr_u_long** routine as its first return parameter. The second return parameter of **callrpc** points to a host machine's pointer that indicates where the return value associated with **unsigned long** is placed. Since the **RUSERSPROC_NUM** function takes no arguments, the **xdr_void** routine is the the argument parameter of **callrpc**.

The **callrpc** routine tries several times to deliver the call message. If it repeatedly gets no reply, **callrpc** returns an error code transported by the User Datagram Protocol (UDP). The **callrpc** routine contains defaults which determine both the number of times **callrpc** tries to deliver a message and the use of UDP as the return value's transport protocol. See the **callrpc** routine in the RPC section of *AIX Operating System Technical Reference* for information on changing these values.

Figure 19-16 shows how to use the **callrpc** routine in a program to determine the number of users logged in to a remote machine.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <utmp.h>
#include <rpcsvc/rusers.h>

main(argc, argv)
     int argc;
     char **argv;
{
     unsigned long nusers;

     if (argc < 2) {
         fprintf(stderr, "usage: nusers hostname\n");
         exit(-1);
     }
     if (callrpc(argv[1],
       RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
       xdr_void, 0, xdr_u_long, &nusers) != 0) {
          fprintf(stderr, "error: callrpc\n");
```

```
            exit(1);
        }
        printf("%d users on %s\n", nusers, argv[1]);
        exit(0) ;
    }
```

Figure 19-16. Using callrpc to Determine Number of Remote Users

19.7.2.2 Using the registerrpc Routine

When network servers are booted, they register the RPC procedures they will handle with the **portmap**., and go into an infinite loop waiting to service remote procedure call requests from clients. The RPC procedures are registered by the portmapper using the **registerrpc** routine. The routine uses six parameters to register an RPC call. The first three parameters register the remote procedure's program, version, and procedure number. The fourth parameter registers the name of the C language procedure required to implement it. The last two parameters identify the data types of the procedure's input and output. (See the Remote Procedure Call section in *AIX Operating System Technical Reference* for information on the **registerrpc** routine.)

The **registerrpc** routine uses the User Datagram Protocol (UDP/IP) as its data transport when it registers an RPC call with a server.

Figure 19-17 shows how to use the **registerrpc** routine in a program to register RPC calls with the portmapper creating an association between a server and RPC program/procedure/version.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <utmp.h>
#include <rpcsvc/rusers.h>

extern char *nuser; /* This routine must be provided by the user*/

main()
{
     registerrpc(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
         nuser, xdr_void, xdr_u_long);
     svc_run();      /* never returns */
     fprintf(stderr, "Error: svc_run returned!\n");
     exit(1);
}
```

Figure 19-17. Using registerrpc to Register RPC Calls with a Portmapper

19.7.3 Using Low-Level RPC

You can change the default values set for RPC routines and manipulate the sockets that transmit remote procedure calls by using the lower level of the RPC library. You should be familiar with sockets and their system calls to program with low-level RPC routines. See *AIX Operating System TCP/IP User's Guide* and *AIX Operating System Technical Reference* for information about sockets.

You can use the lowest level of the RPC library to do the following:

To change to the Transmission Control Protocol/Internet Protoco (TCP/IP) data transport protocol from the default User Datagram Protocol/Internet Protocol (UDP/IP). UDP/IP transports packets of data that contain 8 kilobytes of data or less. TCP/IP transports the data in long streams of data.

To allocate and free memory while serializing (or deserializing message data with XDR (eXternal Data Representation) routines. You cannot free memory in the higher programming levels of RPC.

To perform authentication on a client or server by supplying o verifying credentials.

Subtopics

19.7.3.1 Using the svc_register Routine

Calling the **svc_register** routine illustrates how low level RPC works. Figure 19-18 shows how to write a low-level RPC network application for determining the number of remote users logged in to a server by calling the **svc_register** routine.

```
  #include <stdio.h>
  #include <rpc/rpc.h>
  #include <utmp.h>
  #include <rpcsvc/rusers.h>

  main()
  {

       SVCXPRT *transp;
       int nuser();

       transp = svcudp_create(RPC_ANYSOCK);
       if (transp == NULL){
           fprintf(stderr, "can't create an RPC server\n");
           exit(1);
       }
       pmap_unset(RUSERSPROG, RUSERSVERS);
       if (!svc_register(transp, RUSERSPROG, RUSERSVERS,
                    nuser, IPPROTO_UDP)) {
           fprintf(stderr, "can't register RUSER service\n");
           exit(1);
       }
       svc_run();  /* never returns */
       fprintf(stderr, "should never reach this point\n");
  }
  nuser(rqstp, transp)
       struct svc_req *rqstp;
       SVCXPRT *transp;
  {
       unsigned long nusers;

       switch (rqstp->rq_proc) {
       case NULLPROC:
           if (!svc_sendreply(transp, xdr_void, 0)) {
               fprintf(stderr, "can't reply to RPC call\n");
               exit(1);
           }
           return;
       case RUSERSPROC_NUM:
           /*
            * code here to compute the number of users
            * and put in variable nusers
            */
           if (!svc_sendreply(transp, xdr_u_long, &nusers)) {
               fprintf(stderr, "can't reply to RPC call\n");
               exit(1);
           }
           return;
       default:
           svcerr_noproc(transp);
           return;
```

```
        }
  }
```

Figure 19-18. Using svc_register, a Low-Level RPC Routine

First, the server gets a transport handle, which is used for sending out RPC messages. The **registerrpc** routine uses the **svcudp_create** routine to get a UDP/IP handle. If you require a reliable protocol, you should call the **svctcp_create** routine. If the argument to the **svcudp_create** routine is **RPC_ANYSOCK**, the RPC library creates the socket. If you specify a different socket, it can be either bound or unbound. If you bind the socket to a port, the port numbers of **svcudp_create** and **clntudp_create** must match.

When you make a **clntudp_create** call with an unbound socket, the system gets the port number from the portmap on the machine you are calling. The **portmap** is a daemon that keeps track of the port numbers of all registered RPC services and the servers that can receive them. If the portmap is not running or does not have a port that matches the remote procedure call, the call fails.

Note: You can call the **portmap** directly if you prefer. Search the **<rpc/pmap_prot.h>** header file for the procedure numbers required to call the portmap.

After creating a server transport handle, call the **pmap_unset** routine to erase the last entry for the program number associated with this remote procedure from the portmap tables so the procedure is identified by its new number.

Next, the program number for the **nusers** routine is associated with the procedure **nuser**. The last argument to **svc_register** is usually the protocol that is being used. The default value is **IPPROTO_UDP** so you must change it to **IPPROTO_TCP** if you have created a new TCP/IP socket. The **IPPROTO_UDP** and **IPPROTO_TCP** are constants set in the **<netinet/in.h>** file.

The user routine **nuser** must call and dispatch the appropriate XDR routines based on the procedure number. The user service routine **nuser** serializes the results and returns them to the caller using the **svc_sendreply** routine.

See the RPC Service Routines section of *AIX Operating System Technical Reference* for information on RPC routines.

19.7.3.2 The Client Side of Low-Level RPC

On the client side, you can control the socket used to transport RPC data and the delivery protocol by using the **clnt_call** routine. This contrasts with using the **callrpc** routine in which you have no control over the parameters that set the socket and delivery protocol.

The **clnt_call** routine takes a pointer to a client. You can supply values to the following parameters of **clnt_call**:

Client handl

Procedure number associated with the remote procedur

XDR routine for serializing the parameter

Pointer to the location of the parameter

XDR routine for deserializing the return valu

Pointer to location the return value is place

Time, in seconds, to wait for a reply

The pointer to the client contains the transport protocol value. The **callrpc** routine automatically calls the **clntudp_create** routine to get the pointer. To specify TCP/IP as the delivery protocol, use the **clnttcp_create** routine.

The parameters to the **clntudp_create** routine that can be changed are as follows:

Address and length of the serve

Remote procedure's program numbe

Remote procedure's version numbe

Amount of time the client waits between trie

Pointer to the socket

The number of times the **clnt_call** routine tries to reach the server is determined by dividing the amount of time the client waits between tries by the **clntudp_create** timeout value.

Include the **clnt_destroy** routine to deallocate space associated with the client's handle. It does not close the socket that was passed as a parameter to the **clntudp_create** routine. If there are multiple client handles using the same socket, you can close one handle without destroying the socket used by other handles.

Figure 19-19 shows the low-level RPC code to call the **nusers** service from the server in Figure 19-18 in topic 19.7.3.1.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <utmp.h>
```

```
#include <rpcsvc/rusers.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>

main(argc, argv)
     int argc;
     char **argv;
{
     struct hostent *hp;
     struct timeval pertry_timeout, total_timeout;
     struct sockaddr_in server_addr;
     int addrlen, sock = RPC_ANYSOCK;
     register CLIENT *client;
     enum clnt_stat clnt_stat;
     unsigned long nusers;

     if (argc < 2) {
         fprintf(stderr, "usage: nusers hostname\n");
         exit(-1);
     }
     if ((hp = gethostbyname(argv[1])) == NULL) {
         fprintf(stderr, "can't get addr for %s\n", argv[1]);
         exit(-1);
     }
     pertry_timeout.tv_sec = 3;
     pertry_timeout.tv_usec = 0;
     addrlen = sizeof(struct sockaddr_in);
     bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
         hp->h_length);
     server_addr.sin_family = AF_INET;
     server_addr.sin_port =  0;
     if ((client = clntudp_create(&server_addr, RUSERSPROG,
       RUSERSVERS, pertry_timeout, &sock)) == NULL) {
         clnt_pcreateerror("clntudp_create");
         exit(-1);
     }
     total_timeout.tv_sec = 20;
     total_timeout.tv_usec = 0;
     clnt_stat = clnt_call(client, RUSERSPROC_NUM, xdr_void,
         0, xdr_u_long, &nusers, total_timeout);
     if (clnt_stat != RPC_SUCCESS) {
         clnt_perror(client, "rpc");
         exit(-1);
     }
     clnt_destroy(client);
}
```

Figure 19-19. Low-Level RPC Client Passing an RPC Program using UDP/IP

If you want to make a stream connection, replace the **clntudp_create** routine with the **clnttcp_create** routine. Code the new entry as follows:

```
clnttcp_create (&server_addr, prognum, versnum,
                &socket inputsize, outputsize);
```

Notice that in the **clnttcp_create** routine, there are no **timeout** parameters

to supply, but you do need to supply values for the send and receive buffers. Once the TCP/IP connection is established, all remote procedure calls using the client handle pointed to by this creation routine use this socket. The server that answers a remote procedure call using TCP/IP has its **svcudp_create** routine replaced by the **svctcp_create** routine.

See the RPC Service Routines section of *AIX Operating System Technical Reference* for information about RPC library routines.

A.0 Appendix A. Extended curses Structures

Subtopics
A.1 WINDOW Structure
A.2 PANEL Structure
A.3 PANE Structure

A.1 WINDOW Structure

The ***Extended curses*** library routines use a structure, WINDOW, to hold information about each window that it is working with. Figure A-1 shows the contents of that structure.

```
  struct _win_st
  {
     short    _cury, _curx;
     short    _maxy, _maxx;
     short    _begy, _begx;
     short    _winy, _winx;
     short    _flags;
     short    *_firstch;
     short    *_lastch;
     bool     _clear;
     bool     _leave;
     bool     _scroll;
     ATTR     _csbp;
     NLSCHAR  **_y;
     ATTR     **_a;
     struct   _win_st *_view;
  } ;

  #define     WINDOW struct  _win_st
  #define     _SUBWIN        001
  #define     _ENDLINE       002
  #define     _FULLWIN       004
  #define     _SCROLLWIN     010
  #define     _ISVIEW        040
  #define     _HASVIEW       100
  #define     _STANDOUT      200
  #define     _NOCHANGE       -1
```

Figure A-1. Structure Definition for WINDOW

The variables in this structure perform the following functions:

_cury and _curx
The current (y, x) coordinates for the window. New characters added to the screen are added at this point.

_maxy and _maxx
One more than the maximum values allowed for **_cury** and **_curx**.

_begy and _begx
The starting (y, x) coordinates on the terminal for the window (the home position for the window). The variables **_cury, _curx, _maxy,** and **_maxx** are measured relative to **_begy** and **_begx**, not the home position for the terminal.

winy, _winx
The starting (y, x) coordinates of a viewport within the original window.

_flags

A flag byte that can have one or more of the following values ORed into it:

_SUBWIN
Indicates that the window is a subwindow. The **delwin( )** call checks this flag. If this flag is set, the space for the lines is not freed when the window is deleted.

_END-LINE
Indicates that the end of the line for this window is also the end of a screen.

_FULLWIN
Indicates that this window is a full screen window.

_SCROLLWIN
Indicates that the last character of this screen is at the lower right corner of the terminal. If a character is written to the lower right corner of the terminal, the terminal (hardware) scrolls automatically.

_ISVIEW
Indicates that the window is a viewport window.

_HASVIEW
Indicates that the window has a viewport window in it.

_STANDOUT
Indicates that all characters added to the screen are in standout mode.

***_firstch**

Pointer to the first position (row by row) in the optimization array that was changed. If this pointer contains the value **_NOCHANGE**, then a change was not made to a line since the last time that **refresh( )** changed **curscr**.

***_lastch**

Pointer to the last position (row by row) in the optimization array that has been changed.

_clear

Tells if a clear-screen sequence is to be generated on the next **refresh( )** call. This is only meaningful for screens. The initial clear-screen for the first **refresh( )** call is generated by initially setting **clear** to be TRUE for **curscr**. When this variable is set for the current screen (**curscr**), each **refresh( )** generates a clear screen.

_leave

TRUE if the current (y, x) coordinates and the cursor are to be set to the character position following the last character changed on the terminal, or not moved if there is not a change.

_scroll

TRUE if scrolling is allowed.

_csbp

Current Standout Bit Pattern: The attribute pattern for characters that are written to the window in **standout** mode. See the **_STANDOUT** flag. See Figure 9-7 in topic 9.6 for the patterns that can be combined into this variable.

****_y**
A pointer to an array of lines which describe the terminal. The expression:

```
_y[i]
```

is a pointer to the **i**th line, and:

```
_y [i][j]
```

is the **j**th character on the **i**th line.

****_a**
A pointer to the attribute array space. The expression:

```
_a [i][j]
```

is a pointer to the attribute variable (of data type ATTR) that corresponds to the **j**th character on the **i**th line, represented as **_y [i][j]** in the array specified by the ****_y** field.

struct _win_st *_view
A pointer to the original window from a viewport window.

A.2 PANEL Structure

The **Extended curses** library routines use a structure, PANEL, to hold information about each panel that it is working with. Figure A-2 shows the contents of that structure.

```
#define PANEL   struct Panel

struct  Panel
  {
  short int     p_depth ;
  short int     p_width ;
  short int     orow    ;
  short int     ocol    ;
  char          *title  ;
  char          divty   ;
  char          bordr   ;
  char          fill1   ;
  char          fill2   ;
  PANEL         *p_under;
  PANEL         *p_over ;
  PANE          *fpane  ;
  PANE          *dpane  ;
  PANE          *apane  ;
  WINDOW        *p_win  ;
  int           dfid    ;
  char          plobsc  ;
  char          plmodf  ;
  char          PLfill[6] ;
  }             ;
```

Figure A-2. Structure Definition for PANEL

The variables in this structure perform the following functions:

p_depth Number of rows in panel

p_width Number of columns in panel

orow Origin row (top left)

ocol Origin column

***title** Title string pointer

divty Divide type code

bordr Border flag byte

The following fields are used to relate multiple panels on the display:

***p_under** Next panel in chain under this panel

***p_over** Previous panel in chain over this panel

The following fields are used by the library routines. Do not change

these fields directly:

***fpane**	First pane after divisions
***dpane**	First root pane for div
***apane**	Current active pane
***p_win**	Window struct for panel
dfid	External panel ident
plobsc	Panel obscured flag
plmodf	Panel modified flag
PLfill[6]	Not used

A.3 PANE Structure

The **Extended curses** library routines use a structure, PANE, to hold information about each pane that it is working with. Figure A-3 shows the contents of that structure.

```
#define PANE    struct Pane

struct  Pane
  {
  short int     w_depth ;
  short int     w_width ;
  short int     v_depth;
  short int     v_width ;
  short int     orow    ;
  short int     ocol    ;
  PANE          *vscr   ;
  PANE          *hscr   ;
  PANE          *nxtpn  ;
  PANE          *prvpn  ;
  PANE          *divs   ;
  PANE          *divd   ;
  char          divty   ;
  char          fill1   ;
  short int     divsz   ;
  char          divszu  ;
  char          bordr   ;
  char          fill2   ;
  char          fill3   ;
  WINDOW        *w_win  ;
  WINDOW        *v_win  ;
  int           pnvsid  ;
  PANEL         *hpanl  ;
  PANEPS        *exps   ;
  char          alloc   ;
  char          pnobsc ;
  char          pnmodf ;
  char          PNfill[5]
  }             ;
```

Figure A-3. Structure Definition for PANE

The variables in this structure perform the following functions:

w_depth Rows of data in presentation space for this pane.

w_width Columns of data in presentation space for this pane.

v_depth Rows being shown on the display of this pane including space for borders.

v_width Columns being shown on the display of this pane including space for borders.

orow Top row on panel of view for this pane (including the border).

ocol First column on panel of view for this pane (including the border).

***vscr** Pane to scroll vertically with this pane.

***hscr** Pane to scroll horizontally with this pane.

***nxtpn** Next pane in chain.

***prvpn** Previous pane in chain.

***divs** Next pane that is part of current division specification.

***divd** Start of division of this pane into smaller parts.

divty Division type code that applies to divisions of this pane. May have the following values:

- **Pdivtyv '0'** Divide vertical dimension of this pane. Divisions appear above each other.
- **Pdivtyh '1'** Divide horizontal dimension of this pane. Divisions appear beside each other.

divsz Division size specification:

divszu Division size unit specification that indicates the form for divsz value using one of the following values:

- **Pdivszc '1'** Size is a fixed constant. Fixed constants must be in the range from 1 to the dimension being divided.
- **Pdivszp '2'** Size is a proportional value. Proportional values must be in the range of 1 to 10,000. They represent the number of 10,000ths of the available screen space to assign to the pane.
- **Pdivszf '0'** Size is float. A **float** pane shares an equal amount of the available screen space with all other panes that have the float attribute.

bordr Border flag for this pane.

***w_win** Pointer to pspace window.

***v_win** Pointer to view window.

pnvsid External identifier for this p-pspace and view window.

***hpanl** Pointer to panel that contains this pane.

***exps** Pointer to chain of extra p-spaces for this pane.

alloc An allocation flag that indicates whether ecdfpl allocated the window so that ecrlpl should free it.

pnobsc A flag that indicates that the pane is obscured by an overlayed panel.

pnmodf A flag that indicates the pane was modified.

PNfill[5] Not used.

B.0 Appendix B. PS/2 Printer Support Data Stream

Subtopics
B.1 Using Printers from a Program

B.1 Using Printers from a Program

The printer support of the PS/2 system allows a program to produce output on any installed printer, as long as the program produces an output data stream that conforms to the control and data characters defined in this appendix. Printer support changes that data stream into the specific data stream that the installed printer needs.

Figure B-1 lists the printer control codes to use when printing using the PS/2 printer support. If the printer can perform the function by itself, printer support passes the codes directly to the printer. If the codes do not work on the printer that is installed on your system, printer support performs one of the following actions:

Tries to emulate the control with functions that the installed printe does have

Removes the control from the output stream. The function is no performed.

The three code columns in the table show different representations of the same code, depending on how you enter the code into the data stream:

Control Name This column shows the name of the control character. In many cases this name is the same as the keyboard keys that produce the required ASCII code for the control code.

Hex Code This column shows the hexadecimal representation of the control code.

ASCII Code This column shows the decimal representation of the control code.

Figure B-1. Printer Control Codes

Category	Function Performed	Control Name	Hex Code	ASCII Code
Control:	Provides a null value. Used as a list terminator.	**NUL**	**00**	**0**
	Sounds the	**BEL**	**07**	**7**
	buzzer. (2) Prints the next	**ESC ^**	**1B5E**	**27 94**
	character as a printable character. The next character is a control with an ASCII value of less			

	than 32. Prints more than one character with an ASCII value that is below 32.	**ESC \ m n c**	**1B5C m n c**	**27 92 m n c**
	Clears the printer memory of all data waiting to be printed following the last received line feed. (1)	**CAN**	**18**	**24**
	Sets resolution for raster image print (**n** indicates a string of control bytes). (1)	**ESC [O n**	**1B5B4F n**	**27 91 79 n**
	Performs a printer power-on reset. (1)	**ESC [K**	**1B5B4B 0100 0**	**27 91 75 1 0** **0**
Positioning the Printhead:	Sets back space.	**BS**	**08**	**8**
	Sets horizontal tab.	**HT**	**09**	**9**
	Sets horizontal tabs (**n** is a list of one or more tab positions).	**ESC D n NUL**	**1B44n00**	**27 68 n 0**
	Sets tab stops to power-on settings.	**ESC R**	**1B52**	**27 82**
	Sets line feed.	**LF**	**0A**	**10**

	Sets reverse line feed.	**ESC]**	**1B5D**	**27 93**
	Starts automatic line feed.	**ESC 5 1**	**1B35 1**	**27 53 1**
	Stops automatic line feed.	**ESC 5 0**	**1B35 0**	**27 53 0**
	Provides a carriage return (no line feed).	**CR**	**0D**	**13**
	Provides a vertical tab.	**VT**	**0B**	**11**
	Sets vertical tabs (**n** is a list of tab positions).	**ESC B n NUL**	**1B42 n NUL**	**27 66 n NUL**
Paper Control:	Provides a form feed.	**FF**	**0C**	**12**
	Sets top of forms. (2)	**ESC 4**	**1B34**	**27 52**
	Ignores end of forms. (2)	**ESC 8**	**1B38**	**27 56**
	Respects end of forms. (2)	**ESC 9**	**1B39**	**27 57**
	Sets skip perforation (2) (**n** is lines to skip).	**ESC N n**	**1B4E n**	**27 78 n**
	Stops skip perforation. (2)	**ESC O**	**1B4F**	**27 79**
Formatting the Page	Uses 12 characters-per-inch printing.	**ESC :**	**1B3A**	**27 58**
Image:	Sets 1/8" line	**ESC 0**	**1B30**	**27 48**

spacing.			
Starts **n**/72"	**ESC 2**	**1B32**	**27 50**
line spacing.			
Sets **n**/72" line	**ESC A n**	**1B41 n**	**27 59 n**
spacing.			
Sets page	**ESC C n**	**1B43 n**	**27 67 n**
length. (2) (**n** is lines per page).			
Sets page	**ESC C 0 n**	**1B43 0 n**	**27 67 0 n**
length (2) (**n** is inches per page).			
Sets left and	**ESC X m n**	**1B58 m n**	**27 88 m n**
right margins (**m** and **n** are column numbers).			
Sets top and	**ESC [S m n t1**	**1B5B53 m n t1**	**27 91 83 m n**
bottom margins (**m** and **n** are length of control; **t1 t0** are high/low-order bytes of top margin; **b1 b0** are high/low-order bytes of bottom margin).	**t0 b1 b0**	**t0 b1 b0**	**t1 t0 b1 b0**
Starts	**ESC M 1**	**1B4D 1**	**27 77 1**
automatic line justification.			
Stops automatic	**ESC M 0**	**1B4D 0**	**27 77 0**
line			

	justification. Starts	**ESC P 1**	**1B50 1**	**27 80 1**
	proportional spacing. Stops	**ESC P 0**	**1B50 0**	**27 80 0**
	proportional spacing. Sets print	**ESC [0 n**	**1B5B30 0100 n**	**27 91 48 1 0**
	resolution for draft quality font and **ESC K** image data (**n** indicates the resolution value). (1) Sets	**ESC [L n**	**1B5B4C 0200**	**27 91 76 2 0**
	presentation surface color (**n** indicates the available colors). (1)		**00 n**	**n**
Controlling the Ribbon:	Sets color band 1 (yellow).	**ESC y**	**1B79**	**27 121**
	Sets color band 2 (magenta).	**ESC m**	**1B6D**	**27 109**
	Sets color band 3 (cyan).	**ESC c**	**1B63**	**27 99**
	Sets color band 4 (black).	**ESC b**	**1B62**	**27 98**
	Sets automatic ribbon band shift.	**ESC a**	**1B61**	**27 97**
	Sets the background color for **Text**, **ESC K**, or **ESC [O** printing	**ESC [N n**	**1B5B4E 0100 n**	**27 91 78 1 0**
				n

	(**n** indicates the available colors). (1) Sets the	**ESC [M n**	**1B5B4D 0100 n**	**27 91 77 1 0 n**
	foreground color for **Text**, **ESC K**, or **ESC [O** printing (**n** indicates the available colors). (1) Swaps the	**ESC []**	**1B5B5D 0100 n**	**27 91 93 1 0 n**
	foreground and background printing color (**n**= 0 indicates normal printing; **n**= 1 indicates reversing background and foreground colors). (1)			
Selecting Print Mode:	Starts double-wide printing.	**SO**	**0E**	**14**
	Stops double-wide printing.	**DC4**	**14**	**20**
	Starts double-wide continuous printing.	**ESC W 1**	**1B57 1**	**27 87 1**
	Stops double-wide continuous printing.	**ESC W 0**	**1B57 0**	**27 87 0**
	Starts compressed printing.	**SI**	**0F**	**15**
	Stops	**DC2**	**12**	**18**

compressed printing. Starts	**ESC -1**	**1B2D 1**	**27 45 1**
underline printing. Stops underline	**ESC -0**	**1B2D 0**	**27 45 0**
printing. Starts	**ESC E**	**1B45**	**27 69**
emphasized printing. Stops	**ESC F**	**1B46**	**27 70**
emphasized printing. Starts	**ESC G**	**1B47**	**27 71**
double-strike printing. Stops	**ESC H**	**1B48**	**27 72**
double-strike printing. Starts	**ESC S 0**	**1B53 0**	**27 83 0**
superscript printing. Starts	**ESC S 1**	**1B53 1**	**27 83 1**
subscript printing. Stops	**ESC T**	**1B54**	**27 84**
superscript or subscript printing. Starts color	**ESC [B 1 n**	**1B5B42 0200**	**27 91 66 2 0**
underline, bypassing white space (**n** defines the available color). (1)		**1 n**	**1 n**
Starts	**ESC [B 0 n**	**1B5B42 0200**	**27 91 66 2 0**

	continuous color underline (**n** defines the available color). (1)		**0 n**	**0 n**
	Stops underline.	**ESC [E**	**1B5B450000**	**27 91 69 0 0**
Selecting the Character Set:	Uses PC character set 2.	**ESC 6**	**1B36**	**27 54**
	Uses PC character set 1.	**ESC 7**	**1B37**	**27 55**
	Selects font (**n** specifies the font; varies with printer type).	**ESC I n**	**1B49 n**	**27 73 n**
	Sets graphic set ID (**c** selects graphic set 0, 1 or 2).	**ESC [T 1 0 c**	**1B5B54 1 0 c**	**27 91 84 1 0 c**
Using Bit Image Graphics:	Sets bit graphics normal ((n is a string of control bytes).	**ESC K n**	**1B4B n**	**27 75 n**
	Sets graphics dual-half speed (**n** is a string of control bytes).	**ESC L n**	**1B4C n**	**27 76 n**
	Sets bit graphics dual-normal speed (**n** is a string of control bytes).	**ESC Y n**	**1B59 n**	**27 89 n**
	Sets bit	**ESC Z n**	**1B5A n**	**27 90 n**

graphics high-half speed (**n** is a string of control bytes).			
Sets aspect ratio to 1:1.	**ESC n 1**	**1B6E 1**	**27 110 1**
Sets aspect ratio to 5:6.	**ESC n 0**	**1B6E 0**	**27 110 0**
Moves carriage to home position.	**ESC <**	**1B3C**	**27 60**
Moves right **n**/120.	**ESC d n**	**1B64 n**	**27 100 n**
Moves left **n**/120.	**ESC e n**	**1B65 n**	**27 101 n**
Starts unidirectional printing.	**ESC U 1**	**1B551**	**27 85 1**
Stops unidirectional printing.	**ESC U 0**	**1B550**	**27 85 0**
Sets 7 dot line spacing.	**ESC 1**	**1B31**	**27 49**
Sets graphics line spacing (**n** is the number of 1/216-inch steps).	**ESC 3 n**	**1B33 n**	**27 51 n**
Sets variable space line feed (**n** is the number of 1/216-inch steps).	**ESC J n**	**1B4A n**	**27 74 n**

	Moves vertical presentation down the page in number of dots (**n** indicates how far to move the presentation). (1)	**ESC [U n**	**1B5B55 0100 n**	**27 91 85 1 0 n**
Selecting a Printer:	Selects the printer to accept data. (2)	**DC1**	**11**	**17**
	Deselects the printer so as to not receive data. (2)	**DC3**	**13**	**19**
	Sets initialize function on. (2)	**ESC ? 1**	**1B3F 1**	**27 63 1**
	Sets initialize function off. (2)	**ESC ? 0**	**1B3F 0**	**27 63 0**
	Queries a parallel attached printer for identification. If the device queried is equal to **n**, this printer deactivates the select line. (2)	**ESC Q n**	**1B51 n**	**27 81 n**

(1) Use with the Color Jet Printer.

(2) Do not use these controls when using the print queue.

(3) These controls may not work on the installed printer. Use **passthrough** mode to send these codes to the printer.

C.0 Appendix C. ASCII Characters

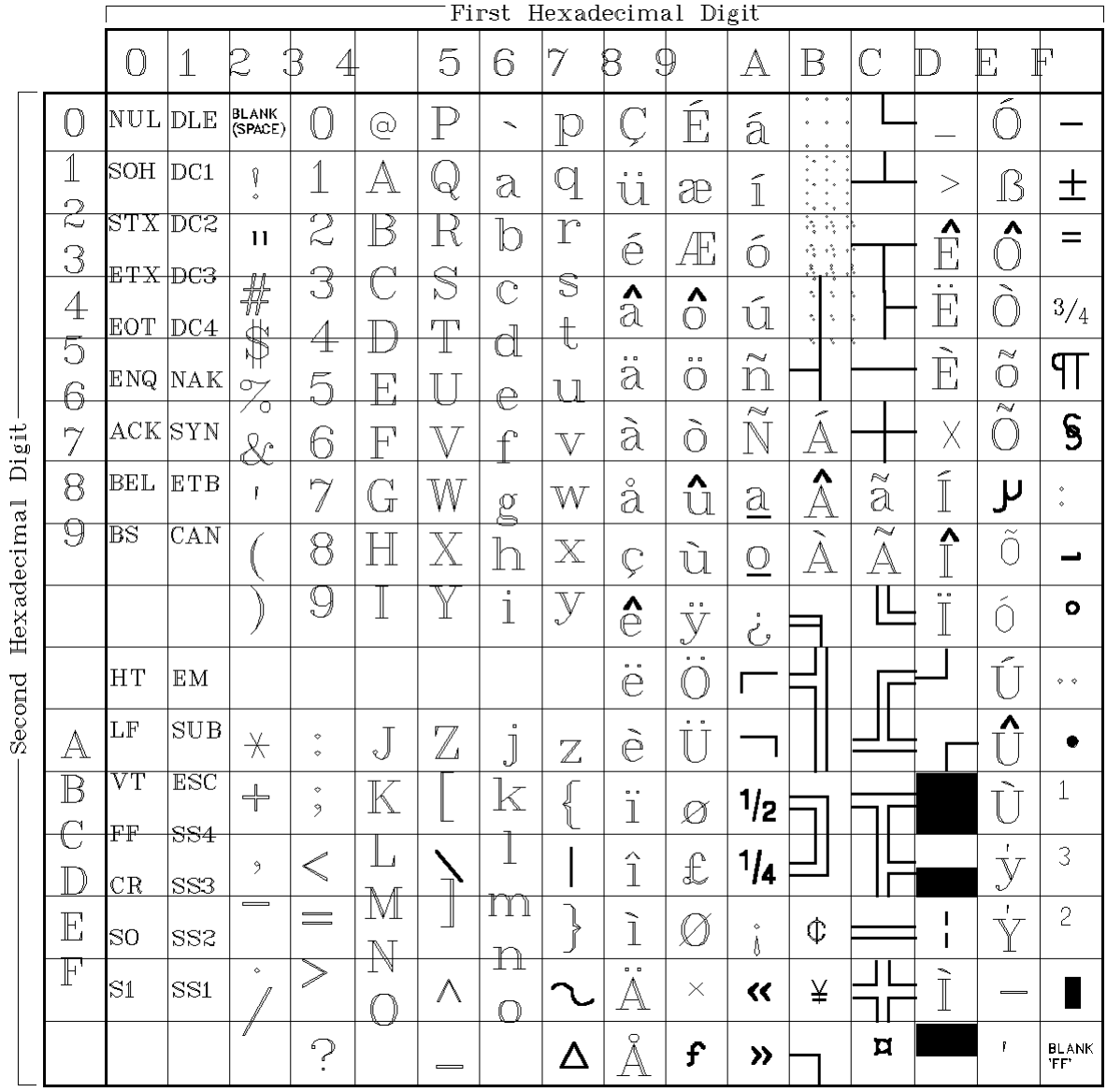


Figure C-1. Code Page 0

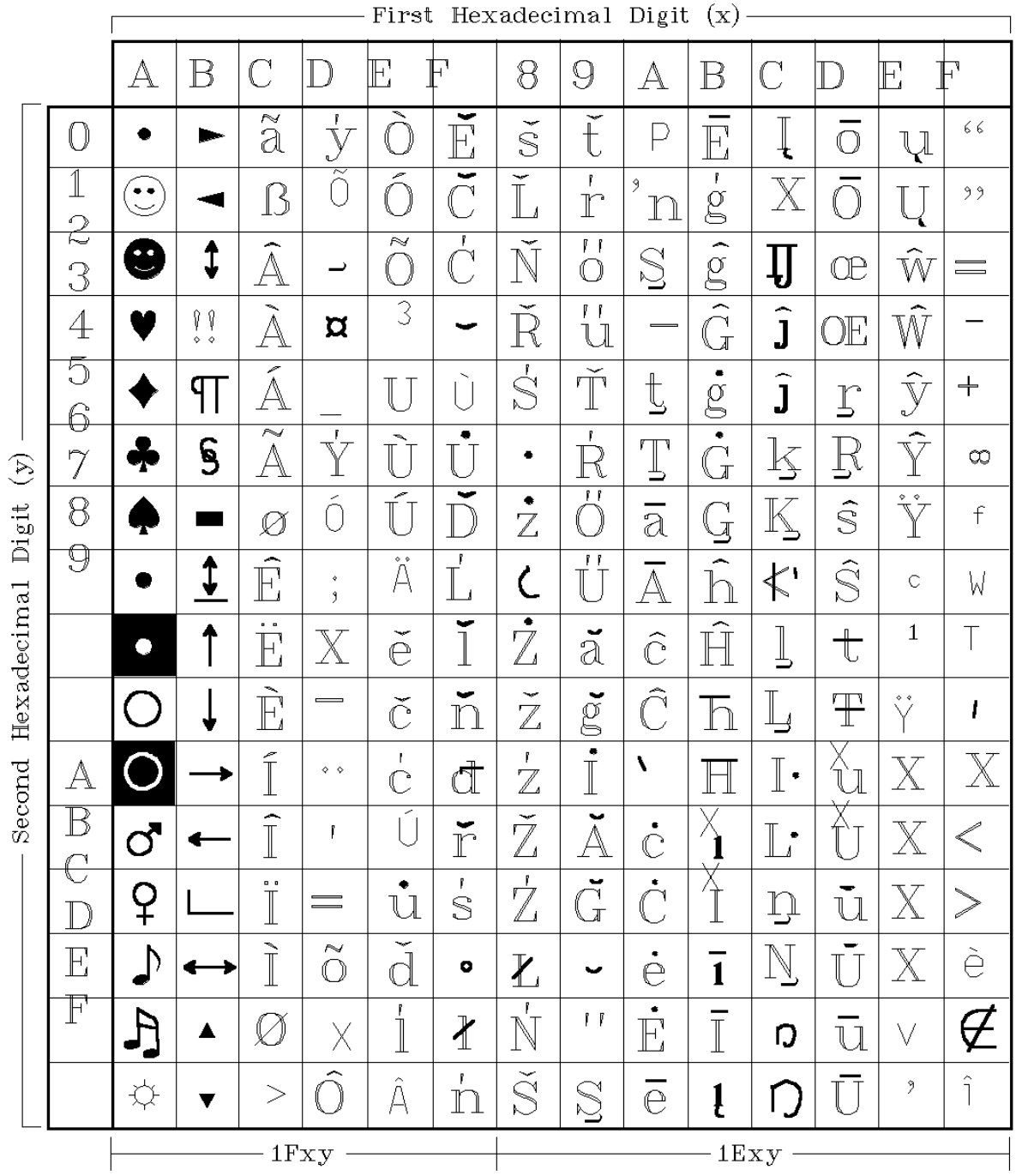


Figure C-2. Code Page 1

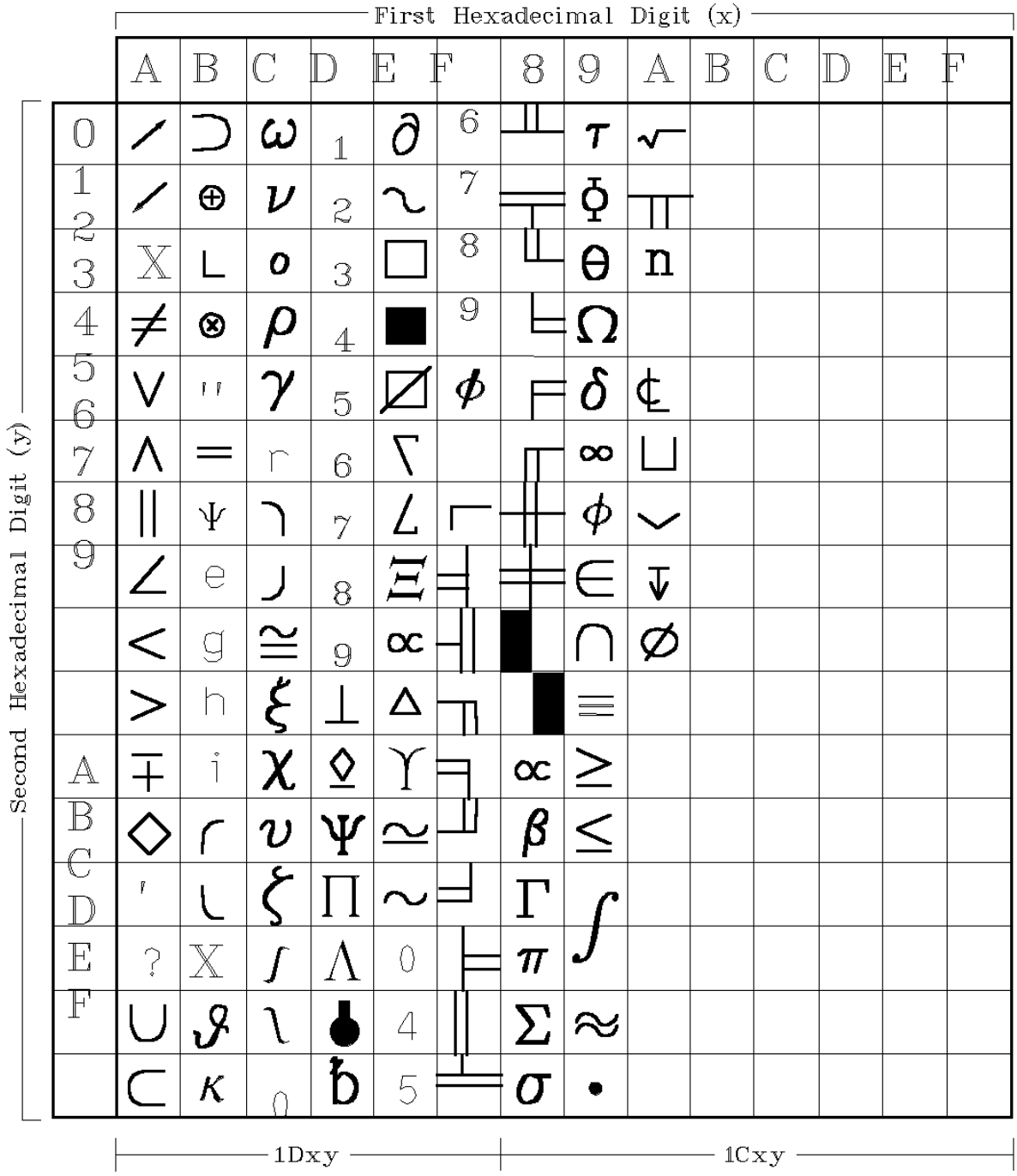


Figure C-3. Code Page 2

D.0 Appendix D. installt Command

The **installt** command usage is documented in the *AIX Operating System Commands Reference*. This section shows menu formats and the format of **installt** tapes.

Subtopics

D.1 Menu Format

The menu selections and their associated **fstore** values are generated by reading the table of contents (TOC) tape file on the media. The format of the TOC is given below. A sample initial menu display may look like Figure D-1.

```
------------------------------------------------------------------------

                                SELECTION MENU

  ID  FSTORE  DESCRIPTION                 ID  FSTORE  DESCRIPTION
   1  asis    Asynch. Terminal Emulation   7  asis    Basic Networking Utility
   2  asis    Graphics Support Library     8  asis    Extended User Support
   3  asis    DOS Server                   9  asis    Message Handler
   4  asis    Learn                       10  asis    Sendmail
   5  asis    CDROM Access Package        11  asis    Sample Programs
   6  asis    Administrative Support      12  asis    Games

  Use 'all' to refer to all packages
  Use 'quit' to exit program


  Enter list of id numbers to install separated by spaces:


------------------------------------------------------------------------
```

Figure D-1. Initial Menu Display

The value "all" can be used to select all items. Items selected for installation are prefixed with '+'. The menu is updated to indicate the user's choices, as in Figure D-2.

```
------------------------------------------------------------------------

                                SELECTION MENU

  ID  FSTORE  DESCRIPTION                 ID  FSTORE  DESCRIPTION
  +1  asis    Asynch. Terminal Emulation   7  asis    Basic Networking Utility
   2  asis    Graphics Support Library     8  asis    Extended User Support
  +3  asis    DOS Server                   9  asis    Message Handler
   4  asis    Learn                       10  asis    Sendmail
  +5  asis    CDROM Access Package        11  asis    Sample Programs
   6  asis    Administrative Support     +12  asis    Games

  Use 'all' to refer to all packages
  Use 'quit' to exit program


  Do you want to change any fstore values (Y/N/HELP)?


------------------------------------------------------------------------
```

Figure D-2. Updated Menu

The **installt** command allows the user to change the **fstore** values used during installation. The **fstore** help text looks like that shown in Figure D-3.

```
  Possible fstore values are:

     asis
     none
     primary
     i386
     i370
     cluster
     all

  Press any key to continue:
```

Figure D-3. Fstore Values

Fstore values are extracted from the **/etc/fstore** file. The 'asis' entry is the normally chosen case and indicates that the **fstore** values associated with files on the media should be used. Figure D-4 shows an example of changing **fstore** values associated with a package.

```
                                SELECTION MENU

  ID  FSTORE  DESCRIPTION                ID  FSTORE  DESCRIPTION
  +1  asis    Asynch. Terminal Emulation  7  asis    Basic Networking Utility
   2  asis    Graphics Support Library    8  asis    Extended User Support
  +3  asis    DOS Server                  9  asis    Message Handler
   4  asis    Learn                      10  asis    Sendmail
  +5  asis    CDROM Access Package       11  asis    Sample Programs
   6  asis    Administrative Support    +12  asis    Games

  Use 'all' to refer to all packages
  Use 'quit' to exit program


  Do you want to change any fstore values (Y/N/HELP)?  Y
  Enter fstore value followed by a list of id numbers (ex:  i386 2 4)
```

Figure D-4. Changing Fstore Values

Fstore values entered must be in the **/etc/fstore** file. The value "all" can be used instead of specifying id numbers to change the **fstore** value of all packages. **installt** gives the user a chance to modify the previous selections. Figure D-5 shows the menu display after answering "no" or entering an empty carriage return.

```
                                SELECTION MENU

  ID  FSTORE  DESCRIPTION                ID  FSTORE  DESCRIPTION
  +1  asis    Asynch. Terminal Emulation  7  asis    Basic Networking Utility
   2  asis    Graphics Support Library    8  asis    Extended User Support
  +3  asis    DOS Server                  9  asis    Message Handler
```

```
   4  asis    Learn                       10  asis    Sendmail
  +5  asis    CDROM Access Package        11  asis    Sample Programs
   6  asis    Administrative Support     +12  asis    Games

  Use 'all' to refer to all packages
  Use 'quit' to exit program


  Are selections correct (Y/N - 'yes' begins installation)?  yes
```

Figure D-5. Menu Display

Answering "no" to the above question clears the selections made by the user, but **fstore** values changed by the user remain changed. Answering "yes" causes the selections to be installed.

D.2 Tape Format

Tapes used with **installt** must be in one of three formats, shown in Figure D-6.

Figure D-6. Tape Formats

All of the images on the tape are in backup format (**installp** images are also in backup format).

Each image must be separated by an end-of-file (EOF) tape mark. EOF tape marks are also found between the BOS and TOC, and TOC and first package. EOF tape marks are automatically placed between the backup images on the media if each image is written with a separate invocation of backup and a no-rewind tape device is used.

The first tape file on IBM LPP Distribution Media contains a backup image of the BOS (also known as opsys).

If the -u flag is specified on the **installt** command, the first tape file on the media is the TOC. Otherwise, the TOC is the second tape file. See below for the format of the TOC.

Additional tape files on the tape contain individual package images in **installp** or **backup** format. These images correspond to entries in the TOC.

D.3 Table of Contents (TOC) Format

The TOC is a backup image containing one file with the path name **./installt_toc**. The TOC is the first tape file if the -u parameter is specified, otherwise it is the second tape file. Entries in the **./installt_toc** file describe the list of packages contained on the tape following the TOC tape file. The entries in the TOC must be in the same order as the images appear following the TOC tape file. The format of an entry in the **./installt_toc** file is shown in Figure D-7.

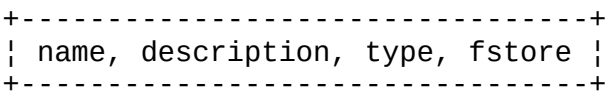


Figure D-7. Format of Entry in ./installt_toc File

The 'name' is usually the same name found in the 'lpp_name' file described in Chapter 13, "Installing and Updating an LPP" in topic 13.0. It can be any name up to 8 characters long.

The 'description' field is usually the description text from the 'lpp_name' file, but can be any description up to 30 characters. The description will be truncated to 27 characters only to fit on the menu display).

The 'type' field determines the type of image and the command used to do the actual installation.

installp	**installp** image, uses **installp** command
i	**installp** image, uses **installp** command
backup	**backup** image, uses **restore** command
b	**backup** image, uses **restore** command

The **fstore** value is passed to the appropriate installation command (i.e. **installp** or **backup**). The **fstore** value is passed to **restore** via the **-F** option and passed to **installp** via the **FRCFSTORE** environment variable. **installp** does not directly use the **FRCFSTORE** environment variable but invokes **/etc/lpp/getfstore**, which sets the value for the **restore** command called from **installp**.

Note: When installing **backup** images that were created with relative file path names, you must be in the directory where the files are to be installed. **installp** images do not require you to be in any particular directory during installation.

Fields in the TOC entries are separated by commas.

Subtopics
D.3.1 Communication Method
D.3.2 Changes to unix.std

D.3.1 Communication Method

If the **installt** command invokes the **installp** command, it communicates with **installp** via a named pipe, called **/tmp/installt.pipe**. If this file exists upon invocation of **installt**, the user is asked whether the file can be deleted and re-created.

D.3.2 Changes to unix.std

If the **unix.std** file is updated during the execution of **installt**, the following message is displayed:

```
The 'unix.std' file has been modified during execution
Enter the command '/etc/reboot' to enable the changes
installed into the kernel.
```

Multiple changes to the **unix.std** file may have taken place during the course of installing packages via **installp**. These changes will not take place until the system is rebooted.

GLOSSARY 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.

action. In **awk**, **lex** and **yacc**, a C language program fragment that defines what the program does when it finds input that it recognizes.

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.

ASCII. See *American National Standard Code for Information Interchange*.

attribute. A characteristic. For example, the attribute for a displayed field could be blinking.

auto carrier return. The system function that places carrier 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.

background process. (1) An activity 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 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.

basename. The last element to the right of a full path name. A file name 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 process records 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. 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 communications, a group of records that is recorded, processed, or sent as a unit. (3) A block is 512 bytes long.

block file. A file listing the usage of blocks on a disk.

block special file. A special file that provides access to a device which is capable of supporting a file system.

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.

bug. A problem in the logic of a program that causes the program to perform differently than expected.

byte. The amount of storage required to represent one character; a byte is 8 bits.

C language. A general-purpose programming language that is the primary language of the AIX Operating System.

call. To activate a program or procedure at its entry point. Compare with *load*.

cancel. To end a task before it is completed.

carrier 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.

channel. One of 32 bits in a table used to represent which event classes are active or inactive. The most significant bit is called **channel 0** and the least significant bit is called **channel 31**.

character. A letter, digit, or other symbol.

character class. Ranges of characters that match a single character.

character display. A display that uses a character generator to display predefined character boxes of images (characters) on the screen. This kind of display can not 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 cannot or do not want to use a file system.

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, for example, the definition is one characteristic of the parent/child relationship. Contrast with *parent*.

close. To end an activity and remove that window from the display.

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. *.symbols. Lower code page P0 (0-127 ordinal) is the ASCII

code segment. See *segment*.

collating sequence. The sequence in which characters are ordered within the computer for sorting, combining, or comparing.

collation. The process of character and string sorting based on alphabetical order and equivalence class.

collation table. Provides an ordered character set and character equivalence classes used by functions.

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.

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 execute a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

command line editing keys. Keys for editing the command line.

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.

compiler. A program that reads program text from a file and changes the programming language statements in that file to a form that the system can understand.

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.

compression. A technique for removing strings of duplicate characters and for removing trailing blanks before transmitting data.

concatenate. (1) To link together. (2) To join two character strings.

concurrent groups. The ability to access files from many groups at the same time.

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*.

constant. A data item with a value that does not change. Contrast with *variable*

control block. A storage area used by a program to hold control information.

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.

copy on reference. A form of memory management provided on AIX/370 whereby a process shares a page of data with other processes until the first attempt to read or write to the page. On the first read or write, the page is copied to give the process its own copy of the data.

copy on write. A form of memory management provided on AIX/PS2 whereby a process shares a page of data with other processes until the first attempt to write to that page. On the first write, the data is copied to a new page at the same virtual memory address before the page of data is changed.

crash. An unexpected interruption of computer service, usually due to a serious hardware or software malfunction.

creation date. The program date at the time a file is created.

current directory. The currently active directory, displayed with the **pwd** command.

current file. In **make**, the file that the **make** command is working with at a given moment. **make** replaces the macro **$*** with the name of the current file.

current line. The line on which the cursor is located.

current screen. In **Extended curses**, the actual image that is currently on the terminal.

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 communications. The transmission of data between computers, or remote devices or both (usually over long distance).

data stream. All information (data and control information) transmitted over a data link.

declaration. A statement in a program that defines how a label is used.

default. A value, attribute, or option 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 standalone 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 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.

diskette. A thin, flexible magnetic plate that is permanently sealed in a protective cover. It can be used to store information copies from the disk or another diskette.

diskette drive. The mechanism 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.

dump. (1) To copy the contents of all or part of storage, usually to an output device. (2) Data that has been dumped.

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.

editor. A program used to enter and modify programs, text, and other

types of documents.

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 set when the user logs in. These variables can be modified later by the user.

equivalence class. A grouping of characters (or character strings) that are considered equal for purposes of collation. For example, many languages place an uppercase character in the same equivalence class as its lowercase form, but some languages distinguish between accented and unaccented character forms for the purpose of collation.

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).

error entry. A data structure containing a header of identifying information plus several bytes of defined data. Error entries are generated by error points and written to an error log file.

error ID. This is part of the data required by an error entry. It is a unique combination of three hexadecimal digits that identifies the component that generated the error entry.

error identifier. A three-character code used to identify error templates and to specify which error entries the error formatter should process. This code is based on the error ID; however, it uses alphanumeric characters instead of hexadecimal digits.

error point. A group of code statements that generates an error entry from within a software program. Error entries are generated when a software or hardware component encounters an error.

error type. One of six categories of errors. The type of an error is determined by the software program that generates the error. When you format an error log, you can specify which types of errors you want to format.

escape character. The backslash character, used to indicate to the shell that the next character is not intended to have the special meaning normally assigned to it by the shell.

event class. A number assigned to a group of trace points that relate to a specific subject or system component. The defined event classes are listed in the trace profile.

exit value. (1) A code sent to either standard output or standard error on completion of the command. (2) 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 character other than a 7-bit ASCII character. An extended character can be a 1-byte code point with the 8th bit set (ordinal 128-255) or a 2-byte code point (ordinal 256 and greater).

fake target name. A control name used in a **makefile** that looks like a target name, but actually tells **make** to perform some operation differently.

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. The smallest component of a record that can be referred to by a name. (2) In **Extended curses**, an area in a presentation space where the program can accept operator input.

FIFO. See *first-in-first-out*.

file. A collection of related data that is stored and retrieved by an assigned name.

file descriptor. A small positive integer that the system uses instead of the file name to identify the file.

file name. The name used by a program to identify a file. See also *label*.

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.

filename. In DOS, that portion of the file name that precedes the extension.

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 rotating 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 always are preceded by a dash.

flattened character. An ASCII character created by translating an extended character to its ASCII equivalent in appearance. Code point information is lost; the character cannot be retranslated to an extended character.

font. A family or assortment of characters of a given size and style.

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.

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 symbol. A symbol defined in one program module, but used in other independently assembled program modules.

graphic character. A character that can be displayed or printed.

group name. A name that uniquely identifies a group to the system.

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.

handler. A software routine that controls a program's reaction to specific external events, such as an interrupt *handler*.

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 file. A text file that contains declarations used by a group of functions or users.

header label. A special set of records on a diskette describing the contents of the diskette.

help. Explanatory information that a program provides.

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 file. A file containing a log of system actions and operator responses.

hole in a file. See *sparse 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.

hook ID. A unique number assigned to a specific trace point. All trace entries include the hook ID of the originating trace point in the trace entry header. Pre-defined trace points use assigned hook IDs ranging from 0 to 299. User-defined trace points can choose hook IDs ranging from 300 to 399.

I/O. See *input/output*.

ID. Identification.

IF expressions. Expressions within a procedure, used to test for a condition.

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 zero or other starting values at the beginning of, or at prescribed points in, the operation of a computer routine.

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.

input. Data to be processed.

input device. Physical devices used to provide data to a computer.

input file. A file opened in the input mode.

input list. A list of variables to which values are assigned from input data.

input-output file. A file opened for input and output use.

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.

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.

key. A unique identifier (of type **key_t**) that names the particular interprocess communications member.

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 user-to-work station dialogue.

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.

kill. An AIX Operating System command that stops a process.

kill character. The character that is used to delete a line of characters entered after the user's prompt.

kilobyte. 1024 bytes.

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 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 filed is in the leftmost position. of the field is in the leftmost position. Contrast with *right-adjust*.

lexical analyzer. A program that analyzes input and breaks it into categories, such as: numbers, letters or operators.

library. A collection of functions, calls, subroutines, or other data.

licensed program product (LPP). Software programs that remain the property of the manufacturer, for which customers pay a license fee.

linefeed. An ASCII character that causes an output device to move forward one line.

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*.

log. To record; for example, to log all messages on the system printer.

log in (v). To sign on at a work station.

log off (v). To sign off at a work 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.

main storage. The part of the processing unit from which programs are run.

mapped file. A file that can be accessed using direct memory operations, rather than having to read it from disk each time it is accessed.

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.

memory. Storage on electronic chips. Examples of memory are random access memory, read only memory, or registers. See *storage*.

memory areas. Arrays of characters in memory.

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) An error indication, or any brief information that a program writes to standard error or a queue. (3) Information sent from one user in a multi-user operating system to another. (4) A general method of communication between two processes.

message queue ID. An identifier assigned to a message queue for use within a particular process. It is similar in use to a *file descriptor* of a file.

message services. A set of routines to help create, update and display messages from a program.

minidisk. A logical division of a fixed disk that may be further subdivided into one or more partitions. See *partitions*.

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.

msqid. See *message queue ID*.

multiprogramming. The processing of two or more programs at the same time.

multivolume file. A diskette file occupying more than one diskette.

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, when coupled with a carriage return, 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. Consists of directly executable machine code. For programs that must be linked, object code consists of relocatable machine code.

octal. A base eight numbering system.

open. 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. A symbol representing an operation to be done.

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 in either the output mode or the extend mode.

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.

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.

paging space. An area on disk that the system uses to store information that is resident in virtual memory, but is not currently being accessed.

pane. In **Extended curses**, an area of the display that shows all or a part of the data contained in a presentation space associated with that pane. A pane is a subdivision of a panel.

panel. In **Extended curses**, a rectangular area on the display consisting of one or more panes that a program can treat as a unit.

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. (1) 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*. (2) Also pertains to a process that has **fork**ed to create one or more child processes.

parent directory. The directory one level above the current directory.

parser. A program that analyzes input and determines what to do with the input.

partition. A logical division of a fixed disk.

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. A complete file name specifying all directories leading to that file.

pattern-matching character. Special characters such as * or ? that can be used in a search pattern. Some are 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 indicating the access permissions. 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.

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 from one process 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.

position. The location of a character in a series, as in a record, a displayed message, or a computer printout.

presentation space. In **Extended curses**, the data and attribute array associated with a window.

primary group. In concurrent groups, the group that is assigned to the files that you create.

print queue. A file containing a list of the names of files waiting to be printed.

printing device. Any printer or device that prints, such as a typewriter-like device or a plotter.

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.

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 ID (PID). A unique number assigned to a process that is running.

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.

queue. A line or list formed by 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.

random access. An access mode in which records can be read from, written to, or removed from a file in any order.

real memory. Memory that is physically present in the system. Contrast with *virtual memory*.

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.

recursion. The process of using a function to define itself.

redirect. To divert data from a process to a file or device to which it would not normally go.

regular expression. A set of characters, metacharacters and operators that define a string or group of strings in a search pattern.

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*.

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. Return to an original value or image. For example, to restore a library from diskette.

right adjust. The process of aligning lines of text at the right margin or tab setting such that the right-most character in the line or file is in the right-most position.

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*.

right margin. The area on a page between the last text character and the right upper edge.

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.

SCCS. See *Source Code Control System*.

SCCS identification. In SCCS, a number assigned to a version of a program to keep track of each version of the program.

scratch file. A file, usually used as a work file, that exists until the program that uses it ends.

screen. (1) See *display screen*. (2) In **Extended curses**, a special type of window that is as large as the terminal screen.

scroll. To move information vertically or horizontally to bring into view information that is outside the display or pane 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.

segment registers. Registers in the system that hold the actual addresses of the memory segments currently in use.

semaphore. A general method of communication between two processes that is an extension of the features of signals.

semaphore ID. An integer that points to a set of semaphores and a data structure that contains information about the semaphores.

semid. See semaphore ID

separator. A character used to separate parts of a command. See *delimiter*.

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.

shared memory. A area of memory that more than one cooperating process can access simultaneously.

shared memory ID. An identifier assigned to the shared segment for use within a particular process. It is similar in use to a *file descriptor* of a file.

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 (there is an AIX shell program and a DOS shell program).

shmid. See shared memory ID.

sign off. To end a session at a display station.

sign on. To begin a session at a display station.

sign-off. The action an operator uses at a display station to end working at the display station.

sign-on. The action an operator uses at a display station to begin working at the display station.

signal. A simple method of communication between two processes.

software. Programs.

sort. To select a particular group of records from a file based upon some criterion. Also, to rearrange some or all of a group of records based

upon items in a particular field of those records.

Source Code Control System (SCCS). A program for maintaining version control for the source files of a developing program.

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.

sparse file. A file that is created with a length greater than the data it contains, leaving empty spaces for future addition of data.

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 ordinary AIX files.

standalone 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.

standard screen. In **Extended curses**, a memory image of the screen that the routines make changes to.

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.

stderr. See *standard error*.

stdin. See *standard input*.

stdout. See *standard output*.

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.

stream. Sequential input or output from an open file descriptor.

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.

subdirectory. A directory contained within another directory in the file system hierarchy.

subprogram. A program invoked by another program. Contrast with *main 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.

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.

superblock. The most critical part of the file system containing information about every allocation or deallocation of a block in the file system.

superuser. The system user with superuser privileges.

superuser privileges. The unrestricted ability to access and modify any part of the operating system associated with the user who manages the system.

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 printout of storage from all active programs (and their associated data) whenever an error stops the system. Contrast with *task dump*.

system profile. A file containing the default values used in system operations.

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.

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 printout of storage from a program that failed (and its associated data). Contrast with *system dump*.

terminal. (1) 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 a computer. See *work station*. (2) In **Extended curses**, a memory image of what the terminal screen currently looks like.

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).

token. (1) The smallest independent unit of meaning as defined by either the parser or the lexical analyzer. A token can contain data, a language keyword, an identifier, or other parts of a language syntax. (2) In M4, any string of letters and digits that **m4** recognizes.

token numbers. Nonnegative integers that represent the names of tokens.

trace. To record data that provides a history of events occurring in the system.

trace entry. A data structure containing a header of identifying information plus up to 20 bytes of defined data. Trace entries are generated by trace points and written to a trace log file.

trace point. A group of code statements that generates a trace entry from within a software program. Trace points are assigned to an event class which can be active or inactive. Trace points with active event classes can generate trace entries.

trace profile. An ASCII file that can be modified to activate or deactivate the various event classes. The trace profile is used by the trace daemon to set up three channel tables that show which event classes are active.

trace template. Used by the trace formatter to determine how the data contained in a trace entry should be formatted. All trace templates are stored in the master template file.

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.

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.

truncate. To shorten a field or statement to a specified length.

typematic key. A key that repeats its function multiple times when held down.

type style. Characters of a given size, style, and design.

Uid. See *user number*.

user ID. See *user number*.

user name. A name that uniquely identifies a user to the system.

user number (Uid). 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.

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.

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 memory. Addressable space that appears to be real memory. From virtual memory, instructions and data are mapped into real memory locations. Contrast with *real memory*.

virtual storage. See virtual memory

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*.

window. In **Extended curses**, a memory image of what a section of the terminal screen looks like at some point in time. A window can be either the entire terminal screen, or any smaller portion down to a single character.

word. A contiguous series of 32 bits (four 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.

working directory. See *current directory*.

Special Characters

Numerics

A

D

T

U