Datalight ROM-DOS™

Developer's Guide

Created: April 2005

Datalight **ROM-DOS™ Developer's Guide**

Datalight, Inc. assumes no liability for the use or misuse of this software. Liability for any warranties implied or stated is limited to the original purchaser only and to the recording medium (disk) only, not the information encoded on it.

U.S. Government Restricted Rights. Use, duplication, reproduction, or transfer of this commercial product and accompanying documentation is restricted in accordance with FAR 12.212 and DFARS 227.7202 and by a license agreement.

THE SOFTWARE DESCRIBED HEREIN, TOGETHER WITH THIS DOCUMENT, ARE FURNISHED UNDER A SEPARATE SOFTWARE OEM LICENSE AGREEMENT AND MAY BE USED OR COPIED ONLY IN ACCORDANCE WITH THE TERMS AND CONDITIONS OF THAT AGREEMENT.

Part Number: 3010-0200-0715

Contents

Chapter 1, Introduction

About ROM-DOS

ROM-DOS is designed to be the best x86 DOS solution available.

Using the fully featured BUILD configuration tool, the embedded system developer can create a DOS kernel which is completely compatible with either standard DOS 6.22 or DOS 7.1 in just a few steps.

In either compatibility mode (6.22 or 7.1), ROM-DOS provides any or all of these features:

- Long Filename Support in the kernel, the command processor, and utilities.
- Boots and/or executes the kernel from a ROM or a disk.
- Change configuration at either compile time or run time (via CONFIG.SYS).
- COMMAND.COM not required to execute the user's application.
- Directly supports LS-120 and other modern hardware without need for drivers.
- Special Century Date handling for older BIOSs.
- Boot menu and dual-boot support.
- International support for 21 countries, both display and keyboard.
- Memory management through the XMS and HIMEM standards.
- Built-in support for a variety of RAM and ROM disks.
- Remote access to disk drives and file transfer using the Zmodem protocol.
- Fully featured DOS in as little as 54kb ROM, 10kb RAM.

ROM-DOS allows a developer to create an embedded system with DOS functionality using Read Only Memory (ROM), flash, or hard disks. Full support for various devices is easily added using device drivers. ROM-DOS works equally well on a system with limited or full hardware resources.

The ROM-DOS SDK includes a variety of utilities for placing an application in ROM, as well as full source for all device drivers. ROM-DOS, provided in library and executable form, offers the following advantages:

- Specifically designed for the embedded system or mobile computing developer.
- All hardware access done exclusively through the BIOS.
- Full source code available.

ROM-DOS Target System Requirements

The target system is the hardware in which ROM-DOS will be running. At a minimum, ROM-DOS requires that the target system include:

- Intel x186 or compatible CPU
- 54k of ROM or disk space for DOS 6.22
- 67k of ROM or disk space for DOS 6.22 with LFN support
- 59k of ROM or disk space for DOS 7.1
- 72k of ROM or disk space for DOS 7.1 with LFN support
- a minimum of 10KB RAM
- as few as eight BIOS calls (depending on configuration)

No special hardware or software, other than that specified above, is required by ROM-DOS. Additional memory may be required for the BIOS and/or the emulated disk drive.

ROM-DOS Development System Requirements

To configure and build a version of ROM-DOS for installation in your target hardware, you'll need Borland TASM and TLINK version 5.2. Compiling source code for device drivers or features such as the mini-command interpreter may also require Borland's compiler BCC. These tools are included in the Datalight Software Developer's Tool Kit (SDTK).

Requesting Technical Assistance

If you encounter a problem in configuring, building, or programming ROM-DOS, please:

- Attempt to resolve the problem by referring to this manual. You can use the table of contents and the index to locate information.
- Check the README.TXT file for any late-breaking changes or additions to the product not covered in the manual.

You can contact Datalight:

- via the web at www.datalight.com
- via email at support@datalight.com
- via telephone at **800.221.6630**

In any communication with Datalight, be sure to include the version and revision information from the original ROM-DOS SDK installation CD-ROM. If you have comments or suggestions about ROM-DOS or documentation, please contact us.

ROM-DOS Basics

Datalight had three goals when designing ROM-DOS: compatibility, flexibility, and affordability. A compatible DOS remains the primary goal for ROM-DOS. If you find a program that does not run correctly under ROM-DOS, but runs under the compatible version of MS-DOS, please contact our Technical Support department.

Whether your hardware is PC-compatible or not; with ROM-DOS your operating system will be compatible. The only requirements for ROM-DOS are RAM, an 80x186 or higher CPU (including the NEC V-series), and in some cases ROM. ROM-DOS can take full advantage of all hardware in your system, including large hard drives, CD-ROM drives, flash memory, and PCMCIA cards.

ROM-DOS performs all interactions with hardware through device drivers. These drivers, provided in full source, work as you would expect from a desktop DOS, whether your system is a small embedded computer, palmtop, or a workstation. ROM-DOS provides a DOS platform on virtually any system.

The following sections describe the various software components that make up a complete system using ROM-DOS. Included is a general description of a DOS-based computer system, as well as some design ideas to aid in hardware selection.

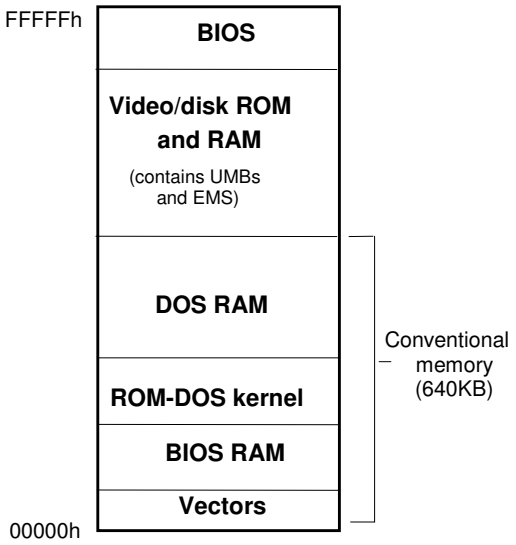
The Major Software Components

At a minimum, ROM-DOS requires a BIOS and a command interpreter to boot the system. The command interpreter can be Datalight's COMMAND.COM (which provides the familiar C:> prompt), or just an application that ROM-DOS runs directly. This program is typically run from a ROM disk on diskless systems or from a floppy/hard disk on systems that have them.

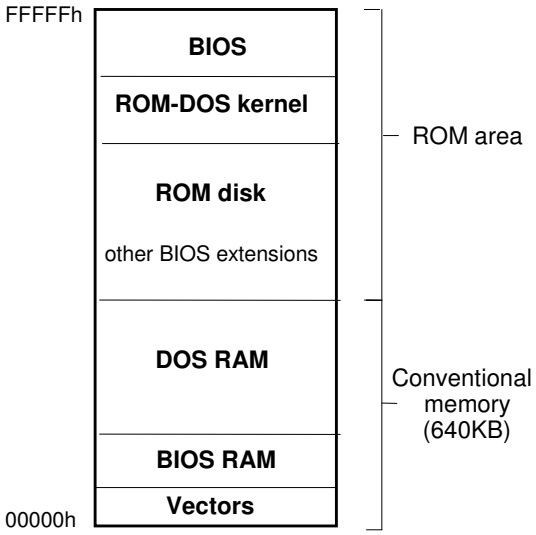
The BIOS gains control as power is applied to the computer, initializes hardware and RAM, and passes control to ROM-DOS. ROM-DOS then determines what hardware support is available through its device drivers, and loads your application or a command interpreter to complete the boot process.

The BIOS always resides in ROM (or some other non-volatile memory). The ROM-DOS kernel can reside and run in ROM, or be loaded from a disk. The command interpreter, COMMAND.COM, or the application program, resides on a disk – either floppy, hard, ROM, RAM, flash (Datalight's FlashFX), CD-ROM or any other disk that DOS can access.

The following illustrations show the locations of the software components in a typical embedded system and in a desktop PC.

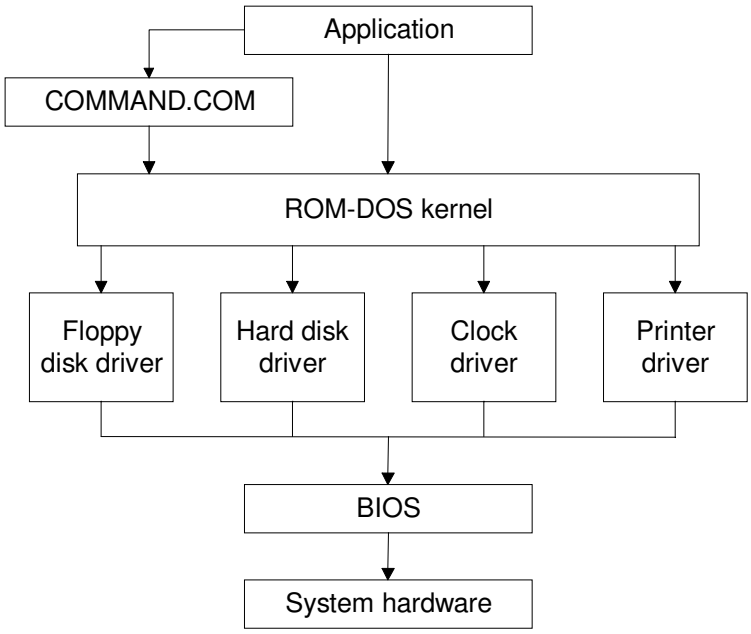


Typical PC System Memory Layout



Typical Embedded System Memory Layout

The following illustration depicts the interaction of software components. The user application communicates with the command interpreter and/or ROM-DOS. These operating system components then communicate with the appropriate device drivers. The device drivers communicate with the BIOS which then makes requests to the system hardware.



Software/Hardware Hierarchy

BIOS

BIOS is an acronym for Basic Input/Output System. All I/O in a system goes through the BIOS, unless the application ties directly to the hardware. The BIOS program is placed in ROM in every desktop computer and any system that can run DOS. It acts as the interface between DOS and the hardware after starting the computer from a power off state. Other functions of the BIOS include initializing any hardware required for the system to run (such as disk drives, the monitor, and so on) and loading DOS.

ROM-DOS performs all I/O operations through its device drivers. The device drivers in turn use BIOS calls. Whether ROM-DOS is writing to the disk or printer, reading from the keyboard, getting the amount of available RAM or time of day, ROM-DOS uses BIOS calls. By using BIOS calls to communicate with the hardware, ROM-DOS does not need to be aware of hardware details.

Most standard PC motherboards include a BIOS specifically configured to operate with the hardware on the motherboard.

ROM-DOS Kernel

The ROM-DOS kernel is the heart of ROM-DOS. The kernel provides file and directory management, input and output through character devices (console, serial port, and printers), along with time and date support. The kernel also provides the ability to load and execute programs, manage memory, and make country-specific information available to applications.

The primary purpose of the kernel is to provide the DOS call services (Int 21h) to programs. The kernel also processes the CONFIG.SYS file during its initialization process and loads the initial

program run by ROM-DOS. This program is by default the command interpreter, COMMAND.COM, although it may be any application program.

The ROM-DOS kernel, like the BIOS, can execute directly from ROM and does not need to copy its code into RAM. Like the command interpreter and other programs, the ROM-DOS kernel can also load from disk and run in RAM. On disk-based systems the ROM-DOS kernel files are named IBMBIO.COM and IBMDOS.COM and are hidden from view. You can list these hidden system files at the command line prompt with the following DIR command.

```
C:\>DIR /as
```

Command Interpreter

The ROM-DOS SDK includes a program known commonly as the command interpreter or shell. Named COMMAND.COM, it is, in most cases the first program loaded by ROM-DOS after the system boots. COMMAND.COM provides the C:\> prompt interface, batch file processing, DIR, ERASE, and other commands. COMMAND.COM allows the user to enter commands using a keyboard and shows the results of the commands on the display.

The primary duties of the command interpreter are: processing the AUTOEXEC.BAT batch file as it starts up, executing internal commands, loading user programs, and executing batch (.BAT) files. The internal commands include DATE, DIR, COPY, TIME, TYPE, and many others.

COMMAND.COM is not the only command interpreter available. The Norton Utilities™, for instance, provides a replacement command interpreter named NDOS. The command interpreter is loaded by an entry in the CONFIG.SYS file such as:

```
SHELL=NDOS.COM
```

Datalight also offers a smaller version of COMMAND.COM called mini-COMMAND. This command interpreter requires only 4KB of ROM or disk space, as opposed to the 43KB of Datalight's full command interpreter.

ROM-DOS boots directly into any compatible program without the need for a command interpreter. Because ROM-DOS systems are sometimes dedicated to a single program, it makes sense to load that program directly, without the additional overhead of COMMAND.COM.

Using a ROM Disk

On systems that have no physical rotating or solid-state disk, a ROM disk can be used to support ROM-DOS. ROM-DOS contains a built-in ROM disk driver along with the more familiar floppy and hard disk drivers. A ROM disk utility, named ROMDISK.EXE, creates a memory image that includes the files you specify. To use the ROM disk utility, specify a directory tree and ROMDISK.EXE creates an image file suitable for your PROM programmer, complete with all the files and the directory structure contained in the directory tree. This ROM disk image can be placed in the same or different ROM as ROM-DOS.

A ROM disk in a diskless system usually contains COMMAND.COM, user applications and data files. From the point of view of ROM-DOS, the ROM disk is nothing more than a fast write-protected floppy disk drive.

Another type of disk is a memory disk. Many MS-DOS developers have used RAMDRIVE.SYS or some other RAM disk equivalent, such as Datalight's VDISK, to speed development. A ROM disk is just a read-only RAM disk. Both RAM and ROM disks are memory disks which, with

some special software (a memory disk device driver), appear to DOS as a conventional disk drive. The directories and files for the disk are located in memory rather than on rotating magnetic media.

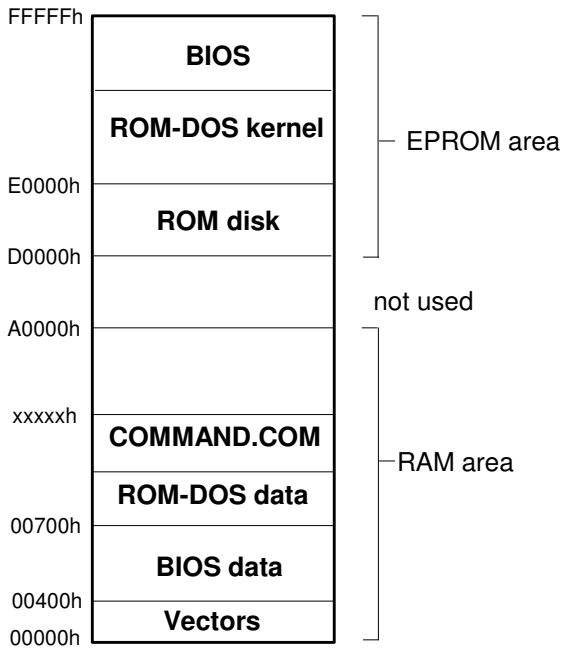
Placing ROM-DOS in a ROM

There are three separate ROM files – the ROM-DOS kernel, ROM disk, and BIOS – that may be programmed into the target system ROM before attempting a boot. When placing ROM-DOS in ROM, its kernel must be configured to execute from ROM. The ROM-DOS SDK contains a utility, BUILD.EXE, to configure the ROM-DOS kernel for a ROM or disk environment.

If the system requires a ROM disk, run the ROMDISK.EXE utility to create the ROM disk image. If the ROM disk is being programmed into ROM, the ROM-DOS ROM disk driver works as-is for ROM disks placed within the 1MB real address space. If your ROM disk is to be placed above the 1MB boundary or your hardware is designed for memory paging, a customized memory disk driver must be created to correctly access the ROM.

If the system requires a custom BIOS, then this ROM must be created and placed on the system.

The following ROM memory map diagram shows a typical memory layout for embedded ROM-DOS system.



ROM-DOS In ROM Memory Map

DOS on a Disk

ROM-DOS does not require any reconfiguration for use on a standard PC platform. A new bootable disk can be made using either the Datalight SYS.COM or FORMAT.COM utilities. The SYS.COM and FORMAT.COM utilities place the hidden system files IBMBIO.COM and IBMDOS.COM on the bootable disk along with the command interpreter COMMAND.COM.

Note: To run SYS.COM or FORMAT.COM, you must boot your system from a disk that contains the hidden system files *or* you can build a ROM-DOS.SYS file (which is equivalent to the hidden system files) as described in 'Chapter 4, Building ROM-DOS' and use it to create a bootable disk.

Using ROM-DOS with Flash Memory

ROM-DOS supports the two basic types of flash memory, PCMCIA cards and on-board flash arrays. ROM-DOS does not directly support PCMCIA cards of any type since this is handled by the BIOS in conjunction with DOS loaded device drivers (generally available from BIOS vendors) called card and socket services. ROM-DOS supports all popular PCMCIA card and socket services device drivers. Onboard flash arrays can be used as a programmable linear memory area or as a disk with read/write capabilities, when used in conjunction with a flash file manager such as Datalight's FlashFX.

ROM-DOS's ROM-disk device driver and ROM-disk building program are suitable for creating an image to place in a linear flash memory and then reading the image as a read-only ROM disk (a ROM disk works with either ROM or flash). The disk image must be programmed into the flash memory using a custom flash loader utility (typically provided by the hardware vendor) or using a PROM programmer capable of programming flash devices.

The use of flash memory differs somewhat from ROM. However, there can be advantages. Flash memory can be less expensive and faster than standard ROM, sometimes even faster than RAM. The high speed of flash is advantageous for running the ROM-DOS kernel from ROM. Some hardware is even set up so that the flash can be reprogrammed on-board while in the field, offering quick and easy updating.

In addition, ROM-DOS includes an ATA device driver named ATA.SYS that supports a variety of ATA cards. Refer to the file ROM-DOS User's Guide for more information on this driver.

What is SOCKETS?

Datalight SOCKETS is an Internet protocol software extension to ROM-DOS that provides a powerful data communication facility whereby embedded systems and users of embedded systems can communicate with other computers (including PCs and mainframes) and their printers.

What does SOCKETS provide?

Datalight SOCKETS provides standard communications applications and the facilities to run custom-written applications which allows you to:

- Run applications on a TCP/IP host system from a remote embedded system.

- Transfer data between an embedded system and TCP/IP hosts.
- Run network aware applications on an embedded system.
- Print to an embedded system from TCP/IP hosts and vice versa.

Datalight Sockets consists of :

- A TSR kernel:
  - Connecting to a physical Ethernet or Token Ring network using a network interface with associated Packet Driver and/or to a point-to-point serial network using standard serial communication ports with or without modem dial in/out.
  - Implementing standard Internet protocols ARP, PPP, LCP, IPCP, IPv6CP, PAP, CHAP MD5, IP, IPv6, ICMP, ICMPv6, IGMP, RIP, UDP, TCP, BOOTP, DHCP and DNS.
  - Providing IP routing support for IPv4.
  - Providing two Application Programming Interfaces (APIs)
  - Providing a Socket Print client
  - Providing a Socket Print Server and LPD Server
  - Optionally keeping MIB II status and statistical information.
- C libraries and source code to access the APIs including a TCP/IP Sockets library implementing the BSD Sockets abstraction. The libraries also support 32 bit applications using a DOS extender.
- A Sockets kernel build program.
- A Sockets configuration program.
- Utility programs to test the network and display the status of the kernel.
- Mail programs in source and binary format.
- Resident servers for FTP, HTTP and Remote Console including a CGI API for serving dynamic web-pages and a Remote Console Java applet to emulate a DOS console of the embedded system on a Java capable browser. Remote Console clients for both DOS and Windows. WebDos, a methodology to enable browser based access to an embedded system, including WebForms for easy browser accessed application development and WebDos Commander for managing the embedded file system.
- An FTP client and DOS and Windows versions of HTTP file GET and PUT utilities.
- Print clients for Socket printing and LPD printing (LPR).
- A resident FTP API to implement FTP client/server functionality in user written programs.
- A resident RFC compliant NETBIOS API allowing file sharing using third party and freeware redirectors and file services.

Chapter 2, About TCP/IP

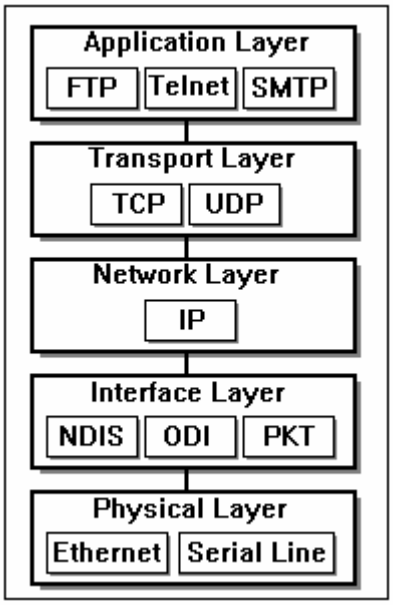
The following sections contain a general description of TCP/IP and provide an introduction to the operation of TCP (Transport Control Protocol) and IP (Internet Protocol) and its components.

Transmission Control Protocol and Internet Protocol, collectively known as TCP/IP, comprise a set of computer data-communication conventions or protocols. These protocols were developed by major users of computer based equipment, principally the U.S. Department of Defense, so that the equipment supplied by different manufacturers could exchange data and information. TCP and IP are only two of the major protocols in a system consisting of many protocols.

TCP/IP Layers

A TCP/IP implementation consists of a series of software layers, where each layer performs specific functions for the layer above and below it. TCP/IP uses four software layers and one physical layer, as follows:

- The Application Layer selects the appropriate service for applications.
- The Transport Layer provides end to end data integrity.
- The Network Layer switches and routes information.
- The Interface Layer transfers units of information to the physical layer.
- The Physical Layer provides transmission onto the network.



Client/Server Model

The most commonly used structure in distributed applications is the client/server model. In this model client applications request services from a server application. The client and server require a set of rules or protocols that must be implemented at both ends of the connection.

The various protocols may act in a Master/Slave role such as Telnet that is used for remote login, or may act in an equally responsive role such as the file transfer protocol (FTP).

File Transfer Protocol (FTP)

The FTP protocol was designed to transfer binary (image) and/or text (ASCII) files between hosts. FTP uses two TCP connections, one for exchanging commands and responses in the form of ASCII strings, the other for the actual data transfers. FTP is implemented in two parts, the Server and the Client. The Server supports multiple, simultaneous, remote users, while the Client provides an interactive or batch interface to the user to perform remote file and directory maintenance and file transfers.

File security is controlled by prompting for the user to specify a name and password that have been configured on the other computer. Provision is made for handling the transfer of files between machines with differing character sets, end of line conventions, etc.

Unlike network file system protocols for *sharing* files, the FTP utility is run only to *transfer* files between systems.

Both FTP client and FTP server applications are supplied with Sockets, including an FTP API for integration with user applications requiring FTP client and server services.

Telnet

Telnet (Network Terminal Protocol) allows users to login on any other host that is connected to the network. These "remote sessions" are started by specifying the host with which a connection is required. Once a connection is established, any local keyboard input is relayed to the remote host and any terminal output from the remote is displayed on the local screen. This is much like a dial-up connection in that the remote system requires log-in and password procedures, as would be encountered in dial-up systems.

At the end of a remote session a logoff command exits the telnet program, and returns the user to the local computer.

A terminal emulation is normally used on top of Telnet on the client side of the connection.

A sample Telnet program is provided with Sockets. A range of Telnet Terminal emulators for Sockets is available from third party vendors.

Mail

The Simple Mail Transfer Protocol (SMTP) allows electronic messages to be sent between hosts on the network.

The SMTP server is used to receive mail and the SMTP client to send mail.

The Post Office Protocol version 3 (POP3) is used to retrieve mail from a mail host. The mail host hosts the POP3 server and a POP3 client resides on a host retrieving the mail.

Hyper Text Transfer Protocol (HTTP)

Hypertext Transfer Protocol is the backbone protocol used by Browsers on the World Wide Web. SOCKETS provides HTTP functionality through an embedded web server and various client applications. Web enabling your device with Datalight SOCKETS will allow easy control of embedded devices from standard desktop web browsers.

The Sockets HTTP server can be extended by user-written programs using an API, to provide for dynamic pages. A powerful system called WebDos is built on this API which allows complete control of an embedded system with Sockets from a standard browser including a "Commander" like interface to manage the embedded system remotely.

Printing

Socket printing is a method of utilising TCP/IP to perform network printing, i.e. printing from any host on the network to a printer attached to any other host. The source of the printing job uses a Print Client to open a TCP connection to a Print Server running on the host that has the destination printer attached to it. The print data is then sent over this connection from the Client to the Server that passes it on to the printer. The end of the print job is signalled by the Client closing the TCP connection. Printer status information may be passed back to Client to signal error conditions such as "Paper out" or "Printer not ready".

Another widely used printing protocol is LPR/LPD. LPR is a print client submitting print jobs to LPD which is a print server. A single LPD print server can handle multiple printers known by name as well as multiple queued jobs.

Sockets can be configured to provide both a built-in LPD and a Socket Print server. A Socket print client or redirector is also provided so that print jobs submitted on one device can be printed on another device or system like a desktop or server machine. Utility type LPR and Socket Print clients are also provided.

Application Programming Interface (API)

An API is a specification of the method an application programmer can use to access services provided by a software module. In the case of a network the API specifies the interface to the network software.

In TCP/IP the idea of a ***Socket*** as the endpoint of a connection is used. A ***socket*** then refers to an abstraction to define the endpoint of a connection as far as the API is concerned. A socket can be created, opened, read, written, closed and deleted in much the same way a file is handled in DOS. The difference is that two sockets must exist, normally on two hosts, before a connection can be made. A read operation on one side must always have a matching write operation on the other side.

Sockets support is provided for the widely used "Socket TCP/IP API" (SAPI) which has been made popular by BSD and WinSock. A lower level API called the Compatible API (CAPI) is also available as well as a proprietary very low level API. Both SAPI and CAPI can be used in a DOS 32-bit environment using DPMI.

Another widely used API is the ***NETBIOS*** API. It differs from the ***SOCKET*** API mainly in the way in which resources on the network are addressed. In the case of the ***SOCKET*** API addressing is done by using IP addresses and port numbers, but for ***NETBIOS*** names are used.

Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) is a second-generation, connection-oriented protocol that corresponds to the Transport layer protocols described in OSI. TCP forms a connection between the workstation and the system with which it intends to communicate. In a network, several systems can communicate across the same network cabling or the same gateways. To ensure that each transmission shares the transmission media equally, transmitted data are broken into manageable pieces known as segments.

TCP is responsible for:

- Breaking data into the appropriate segments.
- Numbering the segments sequentially before sending them.
- Reassembling and verifying the segments at the destination.

The sequential numbers are used to reassemble the segments at the destination. To assist in this procedure, TCP places a header at the beginning of each segment. The header contains the Source Port, the Destination Port, the Sequence number, and a checksum.

A checksum is a mathematical computation of the octets in the segment before it is sent. The same computation is performed at the destination to verify the integrity of the segment data. If the checksum results match, an acknowledgement is sent from the destination to the source. If the checksums do not match, the segment is discarded without an acknowledgement being sent and the source retransmits the segment.

User Datagram Protocol (UDP)

UDP provides a unsequenced, unreliable, connectionless transport service. It can act as an alternative to TCP for applications that do not require the same amount of control. The Domain name protocol, Routing Information protocol and the Simple Network Management protocol all make use of UDP.

2.10. Internet Protocol (IP)

The Internet Protocol is a connectionless network layer protocol and was designed to handle a large number of internetwork connections, for both LAN and WAN applications.

The IP implementation basically addresses and sends the segments. IP relies on the IP address to deliver and receive segments.

Two versions of IP are implemented in Sockets: IPv4 and IPv6. IPv6 is also known as IP Next Generation. IPv4 is widely deployed, while IPv6 deployment is in it's infancy, but it should eventually replace IPv4 completely. The Sockets kernel supports IPv4 or IPv6 or both IPv4 and IPv6.

The IPv4 address is a 32 bit address assigned to an interface on a TCP/IP node. The IP address of each interface must be unique on a network, that is, no two interfaces on nodes anywhere on the network can have the same IP address. If a node has more than one interface, it will also have more than one IP address.

Since a 32 bit address is cumbersome they are generally represented in dotted decimal notation, which separates the four bytes of the address with periods. A typical IPv4 address conversion is as follows:

Type of Format	Example of IP Address
32-bit Binary Format	10000000 01100101 01100110 01100111
Hexadecimal Format	80 65 66 67
Decimal Format	128 101 102 103
Decimal Notation	128.101.102.103

Although the IP address is represented as a single value, it contains two pieces of information:

- The first part of the address is your network identification. All interfaces on the same sub-net have the same prefix.
- The second part of the address is your host identification. No two interfaces on a specific sub-net share the same suffix.

Internet addresses fall into three major addressing classes. The address class that you request should be based on the maximum number of network nodes in your system.

Class	IP Address Range	No of Local Nodes
	0.0.0.1 to 127.255.255.254	1-16,777,214
	128.0.0.1 to 191.255.255.254	1-65,534
	192.0.0.1 to 223.255.255.254	1-254

The table above shows the three Internet address classes with their associated IP address range and the number of local nodes possible per class.

An IPv6 address consists of 128 bits and is represented by up to 8 hexadecimal numbers separated by ":". The longest string of consecutive zero values, can be represented by "::".

Examples:

:: All zeroes address.

::1 seven zero words followed by a word containing 1.

FF80:1::2:345:6789 FF 80 00 01 00 00 00 00 00 00 00 02 03 45 67 89

A node supporting IPv6 may have many IPv6 addresses, but will have at least a Link Local address per interface which is auto-configured using an interface identifier derived from the Mac address of the interface. Additional addresses derived from prefixes supplied by routers in Router Advertisements, will also be auto-configured. Interfaces without a MAC address e.g. a serial link, can be configured for a specific interface identifier or will use a randomly generated interface identifier.

Internet Control Message Protocol (ICMP)

ICMP is used for IPv4 error control and diagnostic. It provides error messages such as:

- destination unreachable
- time to live (ttl) expired
- header problems

ICMP also support echo request and echo reply, better known as a ping operation to test a host for reachability and response time.

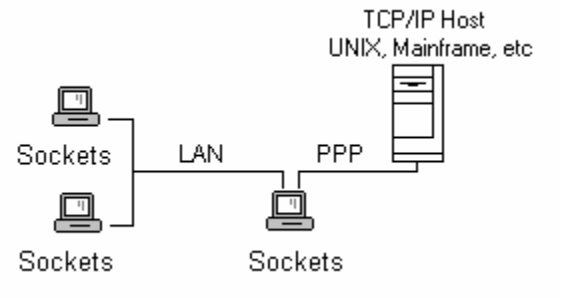
Internet Control Message Protocol version 6 (ICMPv6)

ICMPv6 is used for IPv6 error control and diagnostic as well as for duplicate address detection, automatic address configuration and multicast address listener discovery. It performs for IPv6 the functions performed for IPv4 by ARP, ICMP and IGMP.

Routing

Routers or Gateways are IP nodes on a network that connect more than one network together. This allows a workstation on one network to connect and communicate with a workstation on another network. Routers are often computers that are configured to have more than one network interface. For example when a segment is sent, if the destination network ID matches the source network ID then it is sent directly to the IP address. If the ID's do not match, the segment is sent to a router that knows the ID's of the other connected networks. The segment is forwarded to that router and then ultimately to the IP address. Routing can be applied to Wide Area Networks (WAN) as well as Local Area Networks.

Routing using Sockets is currently only supported for IPv4.



IP Router/Gateway (LAN/WAN)

Internet Gateway Management Protocol (IGMP)

IGMP is used to inform routers of a node's interest in receiving Multicast Datagrams on specific multicast IP addresses in an IPv4 multicasting environment.

Management Information Base version 2 (MIB II)

The Management Information Base is a set of status, control and statistical variables used to measure and control the protocol layers in a TCP/IP stack normally by using a Simple Network Management Protocol (SNMP) agent. An SNMP agent for Sockets is available from a third party vendor. Even without such an agent the optional MIB II statistics can be accessed and displayed by standard utilities and user-written programs.

Routing Information Protocol (RIP)

RIP allows routers to advertise available routes and endpoint workstations and routers to make use of the advertised routes to automatically determine the best route to a destination. Each RIP route has a metric or cost associated with it as well as a limited lifetime so that the network can dynamically adjust to route changes like the failure of a link or router in the network. Both RIP versions 1 and 2 are supported. RIP is only used for IPv4 in Sockets.

Address Resolution Protocol (ARP)

ARP provides a mechanism for a host to map an IPv4 address to the MAC (Ethernet) address of another host on the network. SOCKETS supports ARP for Ethernet and Token Ring controllers.

DHCP ARP is used to ensure that there is no duplicate IP address on a LAN.

Gratuitous ARP is used to inform other hosts on a LAN that the MAC address associated with a specific IP address, may have changed.

Proxy ARP is utilized for gateways. A route can be designated as supporting proxy ARP. When a gateway receives an ARP request for a host and it has a route to reach that host, it responds to the ARP request.

Note: Proxy ARP should be used with care and not in conjunction with RIP. If more than one host responds to an ARP request it may cause system problems.

BOOTP

BOOTP is a UDP/IP based protocol that provides a means to assign an IP address to a booting host dynamically and without user supervision. BOOTP can also supply the net mask, host name, and address of a domain name server. One obvious advantage of this procedure is the centralized management of network addresses, which eliminates the need for per-host unique configuration files. At least one BOOTP server is required on the network.

Dynamic Host Configuration Protocol (DHCP)

DHCP is a UDP/IP based protocol that provides a means to assign the IP address dynamically to a booting host and without user supervision. It can also supply the net mask, host name, address of a domain name server, and other parameters. An advantage of this procedure is the centralized management of network addresses, which eliminates the need for per-host unique configuration files. DHCP provides for address leases and is a better choice than BOOTP.

Point-to-Point Protocol (PPP)

PPP is used on point to point connections e.g. serial links to provide an interface to networking layers including IP. It negotiates configuration settings like header compression at the link level using the Link Control Protocol (**LCP**), authentication using an authentication protocol like Password Authentication Protocol (**PAP**) or Challenge Handshake Authentication Protocol (**CHAP**) and transport layer settings like the IP address using the Internet Protocol Control Protocol (**IPCP**).

PPP is implemented for both IPv4 and IPv6. Currently header compression is only available for IPv4.

Serial Line IP (SLIP)

SLIP uses standard asynchronous lines to transfer IP datagrams. The SLIP provided by SOCKETS is compatible with that used on UNIX systems. Error checking is provided by checksums that are part of IP, TCP and UDP. SLIP is an IPv4 protocol only.

Compressed Serial Line IP (CSLIP)

CSLIP is an enhancement of SLIP by implementing Van Jacobson header compression. CSLIP uses more memory than SLIP but provides better throughput and faster response times, especially on small packets. Like SLIP, CSLIP is an IPv4 protocol only.

Media Support

Various types of media can be used for TCP/IP communication. A brief description of the supported media types follows.

Ethernet and Token Ring

The Packet Driver standard is supported. A Packet Driver is normally supplied by the manufacturer of the network interface controller. Numerous freeware Packet Drivers are also available.

Serial Interface

The standard PC serial interface (COM port) with or without modem dialing/answering is supported by SOCKETS for SLIP, CSLIP or PPP connections.

Modem pool support

When SLIP or CSLIP is used, a proprietary method of supporting modem pools is available. This facility is also known as Multi Destination Driver (MDD) support and can be optionally configured for Sockets. Since SLIP and CSLIP are only used with IPv4, modem pool support is also only available with IPv4.

Alternate Interface support

A proprietary mechanism to determine that traffic via a specific interface is not flowing any more and that an alternate interface should be used, is available for IPv4. This functionality is a standard behavior for IPv6.

References

The formal network standards for the TCP/IP protocol suite is available as a set of documents known as Requests for Comments (RFCs).

Specifications for IP are given in:

- ARPA RFC-791
- MIL-STD-1777

Specifications for TCP are given in:

- ARPA RFC-793
- MIL-STD-1778

Specifications for FTP are given in:

- ARPA RFC-959

Specifications for IPv6 are given in:

- RFC-2460 Internet Protocol, Version 6 (IPv6)
- RFC-2461 Neighbor Discovery for IP Version 6
- RFC-2462 IPv6 Stateless Address Autoconfiguration
- RFC-2463 Internet Control Message Protocol (ICMPv6) for Internet Protocol Version 6
- RFC-2464 Transmission of IPv6 Packets over Ethernet Networks.
- RFC-2472 IP Version 6 over PPP
- RFC-3484 Default Address Selection for Internet Protocol version 6 (IPv6)
- RFC-3513 IP Version 6 Addressing Architecture
- RFC-2710 Multicast Listener Discovery (MLD) for IPv6.
- RFC-2710 Multicast Listener Discovery (MLD) for IPv6.

Chapter 3, Programming Reference

Part of the strength of ROM-DOS and SOCKETS is their accessibility to programmers. The various libraries and APIs documented in this chapter allow the engineer full access to the documented and undocumented interfaces within the DOS kernel and SOCKETS kernel.

Applications developed using these libraries will be compatible with this and future versions of ROM-DOS and SOCKETS, and indeed can be used with other operating systems or compatible application programming interfaces.

Library Use/Linking

Disclaimer

There are other functions in the provided libraries that are not yet documented. Use of these functions is strongly discouraged.

Compilers Supported

These libraries were built in 16-bit DOS mode with Borland C 5.02 and Borland TASM 4.1, and all the library functions have been extensively tested in those environments. Additionally, SOCKETS provides support for the MSVC 1.52 compiler.

Memory Models

The libraries for ROM-DOS, SOCKETS Basic TCP/IP interface, and SOCKETS Advanced TCP/IP interface are provided in Compact, Small, Medium, and Large models.

Library and Header Locations

The libraries for ROM-DOS are located in the LIBS subdirectory of the ROMDOS subdirectory. The include files are similarly located in the INCLUDE subdirectory of the ROMDOS subdirectory.

The libraries for SOCKETS Basic and Advance TCP/IP interfaces are located in the LIB subdirectory of the SOCKETS subdirectory. The include files are similarly located in the INCLUDE subdirectory of the SOCKETS subdirectory.

Header Dependencies

In order to use the ROM-DOS libraries, you must include the appropriate header files. These three files MUST be included first, in this order:

```
DATALGHT.H
DOSTRUCT.H
DOSDEF.H
```

Following this, you may include whichever of the remaining header files you require for library usage. For example, to use the function **AddQuad()** you must include DL64.H.

As with the ROM-DOS libraries, to use the SOCKETS libraries, you must include the appropriate header files. These header files MUST be included in this order:

```
COMPILER.H
CAPI.H
```

If the SOCKETS Advanced TCP/IP interface is being used, then the header files MUST be included in this order:

```
COMPILER.H
CAPI.H
SOCKETS.H
```

Sample Code

ROM-DOS sample code is located within the ROMDOS tree. No added documentation for these examples is provided, but the comments should prove sufficient.

Sample code for SOCKETS is located within the SOCKETS tree.

Contacting Support

If additional information or assistance is needed, please see the section "Requesting Technical Assistance" on page 6.

support@datalight.com

ROM-DOS Libraries

Many modern DOS kernels, such as ROM-DOS, provide support for FAT32, Long Filenames, and LBA bios functions. Unfortunately, most C compilers have little more than DOS.H and BIOS.H, which provide access to the basic 16 bit functions only.

The ROM-DOS libraries fill that gap. The functions documented below provide BIOS level access to modern LBA drives, a standard interface to the Long Filename Functions of Interrupt 0x21, and mathematics using four byte Quad words.

In addition, Datalight has created a set of "Smart" functions, which make the decision of whether to use the Long Filename functions or the standard functions at runtime. For example, instead of **fopen()**, you can now use **SmartCreateOpenFile()**, which will work on a variety of kernels and file systems.

Function Reference

The following sections describe the individual functions of the ROM-DOS libraries.

AddQuad()

The **AddQuad()** function adds an unsigned quad value to the referenced quad word, storing the result in that same location.

C syntax

bool PASCAL AddQuad(uquad * pDstQuad, uquad * pSrcQuad);

Parameters

pDstQuad
Pointer to one summand and the destination quad word structure.

pSrcQuad
Pointer to the other summand.

Return value

Returns one on success, zero on a failure or overflow.

AddQuadLong()

The **AddQuadLong()** function adds an unsigned long value to the referenced quad word, storing the result in that same location. If you are using a constant, this function will show better performance than **AddQuad()**.

C syntax

bool PASCAL AddQuadLong(uquad *pQuad, ulong ulValue);

Parameters

pQuad
Pointer to one summand and the destination quad word structure.

ulValue
An unsigned long value to add to pQuad.

Return value

Returns one on success, zero on a failure or overflow.

ComputeENAMEChecksum()

The **ComputeENAMEChecksum()** function computes the one byte checksum on an ENAME. The ENAME is the eight characters of the short filename, followed by the three character extension with no period. This checksum is stored in the Long Filename directory entries for the file.

C syntax

bool PASCAL ComputeENAMEChecksum(char far * szENAME, uchar far * pucChecksum);

Parameters

szENAME
The zero terminated ENAME string to checksum.

pucChecksum
The far pointer to the location to store the one byte checksum.

Return value

Returns TRUE if the function was successful, otherwise FALSE.

DivideQuadByUnsigned()

The **DivideQuadByUnsigned()** function performs an integer style division of the value in the referenced quad word by the divisor, returning both the quad word result and the unsigned remainder. This function has no return value, as the only failure is a divide by zero. This result will trigger the Divide By Zero interrupt.

C syntax

void PASCAL DivideQuadByUnsigned(uquad *uqpQuad, unsigned uDivisor, unsigned *upRemainder, uquad * puqResult);

Parameters

uqpQuad
Pointer to the dividend, a quad word structure.

uDivisor
The divisor.

upRemainder
Pointer to the remainder of the division.

puqResult
Pointer to the quotient, a quad word structure.

Return value

None.

DlBiosGetDiskStatus()

The **DlBiosGetDiskStatus()** function performs an Interrupt 0x13, function 0x01, then stores the result in the BiosError field.

C syntax

int PASCAL DlBiosGetDiskStatus(int iDrive);

Parameter

iDrive
The physical drive number in the machine, starting at 0. Bit 8 is set for a hard drive, thus the first hard drive is 0x80.

Return value

Returns one on success, zero on a failure. Additionally, any error code returned from Interrupt 0x13 is stored internally, and is accessible from **DlGetBiosError**().

DlBiosGetDriveParameters()

The **DlBiosGetDriveParameters**() function performs an Interrupt 0x13, function 0x08, which returns the parameters of the selected drive in the structure specified.

C syntax

bool PASCAL DlBiosGetDriveParameters(int iDrive, PBDP pbdpParameters);

Parameters

iDrive

The physical drive number in the machine, starting at 0. Bit 8 is set for a hard drive, thus the first hard drive is 0x80.

pbdpParameters

A structure to be filled with appropriate values for drive type, heads, tracks, and sectors per track. See DLINT13.H for a complete description of this packet.

Return value

Returns one on success, zero on a failure. Additionally, the error code returned from Interrupt 0x13 is stored internally, and is accessible from **DlGetBiosError**().

DlBiosReadSectors()

The **DlBiosReadSectors**() function performs an Interrupt 0x13, function 0x02, which reads a number of disk sectors into a provided data area.

C syntax

bool PASCAL DlBiosReadSectors(int iDrive, PBDTP pbdtpPacket);

Parameters

iDrive

The physical drive number in the machine, starting at 0. Bit 8 is set for a hard drive, thus the first hard drive is 0x80.

pbdtpPacket

A structure indicating the read location, number of sectors to read, and the destination pointer. See DLINT13.H for a complete description of this packet.

Return value

Returns one on success, zero on a failure. Additionally, the error code returned from Interrupt 0x13 is stored internally, and is accessible from **DlGetBiosError**().

DlBiosResetDisk()

The **DlBiosResetDisk()** function performs an Interrupt 0x13, function 0x00, in which the disk controller recalibrates the drive heads, causing a seek to track zero.

C syntax

bool PASCAL DlBiosResetDisk(int iDrive);

Parameter

iDrive

The physical drive number in the machine, starting at 0. Bit 8 is set for a hard drive, thus the first hard drive is 0x80.

Return value

Returns one on success, zero on a failure. Additionally, the error code returned from Interrupt 0x13 is stored internally, and is accessible from **DlGetBiosError()**.

DlBiosVerifySectors()

The **DlBiosVerifySectors()** function performs an Interrupt 0x13, function 0x04, which compares sectors in the source pointer with what is read from the drive location specified.

C syntax

bool PASCAL DlBiosVerifySectors(int iDrive, PBDTP pbdtpPacket);

Parameters

iDrive

The physical drive number in the machine, starting at 0. Bit 8 is set for a hard drive, thus the first hard drive is 0x80.

pbdtpPacket

A structure indicating the read location, number of sectors to read and compare, and the source pointer. See DLINT13.H for a complete description of this packet.

Return value

Returns one on success, zero on a failure. Additionally, the error code returned from Interrupt 0x13 is stored internally, and is accessible from **DlGetBiosError()**.

DlBiosWriteSectors()

The **DlBiosWriteSectors()** function performs an Interrupt 0x13, function 0x03, which writes a number of sectors to the disk location specified.

C syntax

bool PASCAL DlBiosWriteSectors(int iDrive, PBDTP pbdtpPacket);

Parameters

iDrive

The physical drive number in the machine, starting at 0. Bit 8 is set for a hard drive, thus the first hard drive is 0x80.

pbdtpPacket

A structure indicating the write location, number of sectors to write, and the source pointer. See DLINT13.H for a complete description of this packet.

Return value

Returns one on success, zero on a failure. Additionally, the error code returned from Interrupt 0x13 is stored internally, and is accessible from **DlGetBiosError**().

dlCheckDOSError

The **dlCheckDOSError()** function provices access to the internally stored return state from the most recent Long Filename function.

C syntax

int PASCAL dlCheckDOSError(void);

Parameters

None.

Return value

This function returns the last error state set by a Long Filename library function.

DlGetBiosError()

The **DlGetBiosError()** function returns the value stored in an internal variable. This corresponds to the table of errors returned by the various Interrupt 0x13 functions.

C syntax

int PASCAL DlGetBiosError(void);

Parameters

None.

Return value

This function returns the last BIOS error state set by a function from the Datalight BIOS library.

dlInWindows()

The **dlInWindows()** function is used to determine if the operating kernel is Windows.

C syntax

bool PASCAL dlInWindows(void);

Parameters

None.

Return value

Returns TRUE if windows is running, otherwise FALSE.

dlIsFat32World()

The **dlIsFat32World()** function is used to determine if the operating system supports DOS 7.1 compatible functions (aka FAT32 functions).

C syntax

bool PASCAL dlIsFat32World(void);

Parameters

None

Return value

Returns TRUE if we are in a DOS 7.1 (FAT32) environment.

DlLbaGetDriveParameters()

The **DlLbaGetDriveParameters()** function performs an Interrupt 0x13, function 0x48. LBA is supported on most modern BIOSes, and is required for drives larger than 8 gigabytes.

C syntax

bool PASCAL DlLbaGetDriveParameters(int iDrive, PEBDPT pebdptParameters);

Parameters

iDrive

The physical hard drive number in the machine, starting at 0x80, since bit 8 is set for a hard drive.

pebdptParameters

A structure, the standard Extended Bios Device Parameter Table, to contain the returned values. See DLINT13.H for a complete description of this packet.

Return value

Returns one on success, zero on a failure. Additionally, the error code returned from Interrupt 0x13 is stored internally, and is accessible from **DlGetBiosError()**.

DlLbaReadSectors()

The **DlLbaReadSectors()** function performs an Interrupt 0x13, function 0x42, which reads sectors from an LBA drive, using the LBA sector offset instead of the cylinder, head, and track combination. LBA is supported on most modern BIOSes, and is required for drives larger than 8 gigabytes.

C syntax

bool PASCAL DlLbaReadSectors(int iDrive, PDAP pdapAddressPacket);

Parameters

iDrive

The physical drive number in the machine, starting at 0. Bit 8 is set for a hard drive, thus the first hard drive is 0x80.

pdapAddressPacket

A structure indicating the read location, number of sectors to read, and the destination pointer. See DLINT13.H for a complete description of this packet.

Return value

Returns one on success, zero on a failure. Additionally, the error code returned from Interrupt 0x13 is stored internally, and is accessible from **DlGetBiosError()**.

DlLbaVerifySectors()

The **DlLbaVerifySectors()** function performs an Interrupt 0x13, function 0x44, which reads sectors from an LBA drive, using the LBA sector offset instead of the cylinder, head, and track combination. These sectors are compared with the source data pointed to by the structure, for verification. LBA is supported on most modern BIOSes, and is required for drives larger than 8 gigabytes.

C syntax

bool PASCAL DlLbaVerifySectors(int iDrive, PDAP pdapAddressPacket);

Parameters

iDrive

The physical drive number in the machine, starting at 0. Bit 8 is set for a hard drive, thus the first hard drive is 0x80.

pdapAddressPacket

A structure indicating the verify location, number of sectors to read and compare, and the source pointer. See DLINT13.H for a complete description of this packet.

Return value

Returns one on success, zero on a failure. Additionally, the error code returned from Interrupt 0x13 is stored internally, and is accessible from **DlGetBiosError()**.

DlLbaWriteSectors()

The **DlLbaWriteSectors()** function performs an Interrupt 0x13, function 0x43, which writes sectors to an LBA drive, using the LBA sector offset instead of the cylinder, head, and track combination. LBA is supported on most modern BIOSes, and is required for drives larger than 8 gigabytes.

C syntax

bool PASCAL DlLbaWriteSectors(int iDrive, PDAP pdapAddressPacket);

Parameters

iDrive

The physical drive number in the machine, starting at 0. Bit 8 is set for a hard drive, thus the first hard drive is 0x80.

pdapAddressPacket

A structure indicating the write location, number of sectors to write, and the source pointer. See DLINT13.H for a complete description of this packet.

Return value

Returns one on success, zero on a failure. Additionally, the error code returned from Interrupt 0x13 is stored internally, and is accessible from **DlGetBiosError()**.

DlSmartLbaGetDriveParameters()

The **DlSmartLbaGetDriveParameters()** function determines at runtime whether to use the LBA get drive parameters call or the standard BIOS function. The results of either are stored in a common structure, which is used for other Smart functions.

C syntax

bool PASCAL DlSmartLbaGetDriveParameters(int iDrive, PLSIP plsipInfo);

Parameters

iDrive

The physical drive number in the machine, starting at 0. Bit 8 is set for a hard drive, thus the first hard drive is 0x80.

plsipInfo

A structure containing the number of sectors on the drive, drive type, and also the standard cylinder, head, and track geometry.

Return value

Returns one on success, zero on a failure. Additionally, any error code returned from Interrupt 0x13 is stored internally, and is accessible from **DlGetBiosError()**.

DlSmartLbaReadSectors()

The **DlSmartLbaReadSectors()** function determines at runtime whether to use the LBA read sectors call or the standard BIOS function. The results of either are stored in the appropriate buffer.

C syntax

bool PASCAL DlSmartLbaReadSectors(PLSIP plsipInfo, PLSTP plstpTransfer);

Parameters

plsipInfo

A structure containing the number of sectors on the drive, drive type, and also the standard cylinder, head, and track geometry. See DLINT13.H for a complete description of this structure, which is returned by **DlSmartLbaGetDriveParameters()**.

plstpTransfer

A structure containing transfer information, such as the read location, number of sectors to read, and the destination pointer. See DLINT13.H for a complete description of this packet.

Return value

Returns one on success, zero on a failure. Additionally, any error code returned from Interrupt 0x13 is stored internally, and is accessible from **DlGetBiosError()**.

DlSmartLbaVerifySectors()

The **DlSmartLbaVerifySectors()** function determines at runtime whether to use the LBA read sectors call or the standard BIOS function. The results of either are verified against data stored in the buffer.

C syntax

bool PASCAL DlSmartLbaVerifySectors(PLSIP plsipInfo, PLSTP plstpTransfer);

Parameters

plsipInfo

A structure containing the number of sectors on the drive, drive type, and also the standard cylinder, head, and track geometry. See DLINT13.H for a complete description of this structure, which is returned by **DlSmartLbaGetDriveParameters()**.

plstpTransfer

A structure containing transfer information, such as the read location, number of sectors to read, and the source pointer for the compare. See DLINT13.H for a complete description of this packet.

Return value

Returns one on success, zero on a failure. Additionally, any error code returned from Interrupt 0x13 is stored internally, and is accessible from **DlGetBiosError()**.

DlSmartLbaWriteSectors()

The **DlSmartLbaWriteSectors()** function determines at runtime whether to use the LBA write sectors call or the standard BIOS function. The results of either are stored in the appropriate buffer.

C syntax

bool PASCAL DlSmartLbaWriteSectors(PLSIP plsipInfo, PLSTP plstpTransfer);

Parameters

plsipInfo

A structure containing the number of sectors on the drive, drive type, and also the standard cylinder, head, and track geometry. See DLINT13.H for a complete description of this structure, which is returned by **DlSmartLbaGetDriveParameters()**.

plstpTransfer

A structure containing transfer information, such as the write location, number of sectors to write, and the source pointer. See DLINT13.H for a complete description of this packet.

Return value

Returns one on success, zero on a failure. Additionally, any error code returned from Interrupt 0x13 is stored internally, and is accessible from **DlGetBiosError()**.

DriveSupportsLFNs()

The **DriveSupportsLFNs()** function tests whether the specified drive supports Long Filenames. All the physical drives in a specific LFN kernel will support LFNs, but certain Network or ATAPI devices (such as CD-ROM) might not.

C syntax

bool DriveSupportsLFNs(uchar ucDrive);

Parameter

ucDrive

The DOS drive parameter (0=the current drive, 1=A:, 2=B:, etc.).

Return value

Returns TRUE if the drive supports LFNs, otherwise FALSE.

GetSmartFindLFNAddress()

The **GetSmartFindLFNAddress()** function returns the address of the Long File Name string from within an internal structure. This value should be copied immediately, as it will go stale with many subsequent FindFirst and FindNext calls.

C syntax

void PASCAL GetSmartFindLFNAddress(char ** pszLFN);

Parameter

pszLFN

A location to contain the returned character pointer.

Return value

Returns the address of the LFN from an internal LFN find structure.

LFNChangeDirectory()

The **LFNChangeDirectory()** function will change to a given directory using Long Filename paths. This function also adjusts for a bug in the current "Windows NT" implementation of the LFN functions, which change the current drive on this call.

C syntax

bool PASCAL LFNChangeDirectory(char szLongName[]);

Parameter

szPathName

The zero terminated argument string for the desired path.

Return value

Returns TRUE if the change directory was successful, otherwise FALSE. Any error will be available through **dlCheckDOSError()**.

LFNCreateOpenFile()

The **LFNCreateOpenFile()** function is used to create or open a Long Filename.

C syntax

bool PASCAL LFNCreateOpenFile(int iModeFlags, int iAttr, int iAction, char szLongName[], int iAliasHint, int *pHandle, int *pActionTaken);

Parameters

iModeFlags

Various flags for the file open mode, as defined in DLLFN.H:

```
#define OPEN_ACCESS_READONLY            0
#define OPEN_ACCESS_WRITEONLY           1
#define OPEN_ACCESS_READWRITE           2
#define OPEN_ACCESS_RO_NOMODLASTACCESS  4
#define OPEN_SHARE_COMPATIBLE           0
#define OPEN_SHARE_DENYREADWRITE        0x10
#define OPEN_SHARE_DENYWRITE            0x20
#define OPEN_SHARE_DENYREAD             0x30
#define OPEN_SHARE_DENYNONE             0x40
#define OPEN_FLAGS_NOINHERIT            0x80
#define OPEN_FLAGS_NO_BUFFERING         0x100
```

```
#define OPEN_FLAGS_NO_COMPRESS          0x200
#define OPEN_FLAGS_ALIAS_HINT           0x400
#define OPEN_FLAGS_NOCRITERR            0x2000
#define OPEN_FLAGS_COMMIT               0x4000
```

iAttr

The desired attribute for the resulting file, as defined in DLLFN.H:

```
#define A_NORMAL          0x00
#define A_READONLY        0x01
#define A_HIDDEN          0x02
#define A_SYSTEM          0x04
#define A_VOLUME          0x08
#define A_SUBDIR          0x10
#define A_ARCHIVE         0x20
#define A_ALLDIR          0x17
```

iAction

A flag used to indicate whether a Create, Open or Truncate is desired, as defined in DLLFN.H:

```
#define FILE_CREATE       0x10
#define FILE_OPEN         1
#define FILE_TRUNCATE     2
```

szLongName

The zero terminated Long Filename.

iAliasHint

If the proper bit flag is set in iModeFlags (OPEN_FLAGS_ALIAS_HINT), this value will be used (if possible) to create the short name alias.

pHandle

The returned file handle which can be used to further access the file.

pActionTaken

The returned action taken by the function; whether it Opened, Created or Truncated a file. For useful #defines, look in the usual spot.

Return value

Returns TRUE if the file is opened or created, otherwise FALSE. Any error will be available through **dlCheckDOSError()**.

LFNDeleteFiles()

The **LFNDeleteFiles()** function removes a Long Filename from the drive and/or path specified in the szLongMask. The search attributes are not used unless wild cards are OK.

C syntax

bool PASCAL LFNDeleteFiles(int iAttrs, char szLongMask[], bool bWildOK);

Parameters

iAttr

The "must match" and "search" attributes, as defined in DLLFN.H:

```
#define MUST_MATCH_ATTR(a) ((int)(a)<<8)
#define SEARCH_ATTR(a) ((int)(a)&0xFF)
#define A_NORMAL          0x00
#define A_READONLY        0x01
```

```
#define A_HIDDEN          0x02
#define A_SYSTEM          0x04
#define A_VOLUME          0x08
#define A_SUBDIR          0x10
#define A_ARCHIVE         0x20
#define A_ALLDIR          0x17
```

szLongMask

The zero terminated Long Filename or Long Filename mask argument.

bWildOK

Set to TRUE if wildcards are acceptable. If FALSE, the search attributes are also ignored.

```
#define DEL_NOWILD        0
#define DEL_WILD          1
```

Return value

Returns TRUE if the function was successful, otherwise FALSE. Any error will be available through **dlCheckDOSError()**.

LFNEndArg()

The **LFNEndArg()** function moves the pointer to the end of the current argument. This function is aware of international characters and quoted strings, treating the latter as one argument.

C syntax

char * PASCAL LFNEndArg(char *szArg);

Parameter

szArg

A pointer to the zero terminated argument string.

Return value

Returns a pointer to the first space after the argument, or the terminating NULL in the case of a final argument which is not followed by spaces.

LFNExtendedGetSetAttr()

The **LFNExtendedGetSetAttr()** function performs a number of tasks, based on the value passed in the iAction field. These include getting and setting various file attributes, dates, and times. An additional action returns the filesize.

C syntax

bool PASCAL LFNExtendedGetSetAttr(int iAction, FileTime *pTime, int *pAttr, char szLongName[], FileDate *pDate, int *pMilli, long *pFileSize);

Parameters

iAction

The action that the function is to take, as defined in DLLFN.H:

```
#define EXT_ACT_GET_ATTR                    0
```

```
#define EXT_ACT_SET_ATTR                        1
#define EXT_ACT_GET_PHYSICAL_SIZE               2
#define EXT_ACT_SET_LAST_WRITE_DATE_TIME        3
#define EXT_ACT_GET_LAST_WRITE_DATE_TIME        4
#define EXT_ACT_SET_LAST_ACCESS_DATE            5
#define EXT_ACT_GET_LAST_ACCESS_DATE            6
#define EXT_ACT_SET_CREATION_DATE_TIME          7
#define EXT_ACT_GET_CREATION_DATE_TIME          8
```

pTime
An unsigned integer containing the time, which is used for some actions.

pAttr
A bitfield containing the file attributes, where are used for some actions.

szLongName
The zero terminated Long Filename.

pDate
An unsigned integer containing the date, which is used for some actions.

pMilli
The value representing the milliseconds portion of the file time, in numbers of 10 millisecond units (i.e. 13 = 130 milliseconds).

pFileSize
A pointer to the unsigned long field to receive the files size from some actions.

Return value

Returns TRUE if the function was successful, otherwise FALSE. Any error will be available through **dlCheckDOSError**().

LFNGetCreateTimeDate()

The **LFNGetCreateTimeDate**() function returns the creation date and time of the specified file. This data is returned from the Long Filename directory structure, and so may not be available from conventional short-name files.

C syntax

bool LFNGetCreateTimeDate(int nHandle, FileTime *pCreateTime, FileDate *pCreateDate, int *pCreateMilli);

Parameters

nHandle
The DOS file handle

pCreateTime
The location to store the DOS time structure for the file. See DLLFN.H for a complete definition of this structure.

pCreateDate
The location to store the DOS date structure for the file. See DLLFN.H for a complete definition of this structure.

pCreateMilli

The location to store the value representing the milliseconds portion of the file time, in numbers of 10 millisecond units (i.e. 13 = 130 milliseconds).

Return value

Returns TRUE if the function was successful, otherwise FALSE. Any error will be available through **dlCheckDOSError**().

LFNGetCurrentDirectory()

The **LFNGetCurrentDirectory**() function returns the current working directory of the selected drive.

C syntax

bool PASCAL LFNGetCurrentDirectory(int nDrive, char szLongName[]);

Parameters

nDrive

The DOS drive parameter (0=the current drive, 1=A:, 2=B:, etc.).

szLongName

The location to receive the zero terminated Long Filename path.

Return value

Returns TRUE if the function was successful, otherwise FALSE. Any error will be available through **dlCheckDOSError**().

LFNGetFullPath()

The **LFNGetFullPath**() function takes the passed short, long, mixed or relative path, and returns the short, long, or mixed version of the path. Depending on the flags, it can also expand the true path of a SUBSTed drive.

C syntax

bool PASCAL LFNGetFullPath(int iFlags, char szSrcLongName[], char szDstLongName[]);

Parameters

iFlags

Various flags indicating the type of path data to return, as defined in DLLFN.H:

```
#define FULLPATH_NOSUBST          0
#define FULLPATH_SUBST            0x8000
#define FULLPATH_DEFNAME          (0 | FULLPATH_SUBST)
#define FULLPATH_SHORTNAME        (1 | FULLPATH_SUBST)
#define FULLPATH_LONGNAME         (2 | FULLPATH_SUBST)
```

szSrcLongName

The zero terminated source path, with or without Long Filenames and relative components.

szDstLongName
The location to store the zero terminated result of the function.

Return value
Returns TRUE if the function was successful, otherwise FALSE. Any error will be available through **dlCheckDOSError()**.

LFNGetLastAccessDate()

The **LFNGetLastAccessDate()** function returns the last accessed date of the specified file. This data is returned from the Long Filename directory structure, and so may not be available from conventional short-name files.

C syntax
bool LFNGetLastAccessDate(int nHandle, FileDate *pLastAccessDate);

Parameters
nHandle
The DOS file handle

pLastAccessDate
The location to store the DOS date structure for the file. See DLLFN.H for a complete definition of this structure.

Return value
Returns TRUE if the function was successful, otherwise FALSE. Any error will be available through **dlCheckDOSError()**.

LFNGetVolumeInformation()

The **LFNGetVolumeInformation()** function returns some specific information about the file system on the selected volume, including filename lengths, pathname lengths, and other useful flags.

C syntax
bool PASCAL LFNGetVolumeInformation(int iBufSize, char szRootName[], char szFileSystemName[], int *pFlags, int *pMaxName, int *pMaxPath);

Parameters
iBufSize
The size of the szFileSystemName buffer. 32 bytes should be sufficient.

szRootName
The root of the selected drive, in fairly specific format (e.g. "C:\").

szFileSystemName
A buffer to receive the zero terminated name of the filesystem (e.g. "FAT", "NTFS" or "CDFS")

pFlags

The file system description flags, defined in DLLFN.H:

```
#define FS_CASE_SENSITIVE         0x0001
#define FS_CASE_IS_PRESERVED      0x0002
#define FS_UNICODE_ON_DISK        0x0004
#define FS_LFN_APIS               0x4000
#define FS_VOLUME_COMPRESSED      0x8000
```

pMaxName

The location where the maximum length of a filename is returned.

pMaxPath

The location where the maximum length of a filepath is returned.

Return value

Returns TRUE if the function was successful, otherwise FALSE. Any error will be available through **dlCheckDOSError**().

LFNMakeDirectory()

The **LFNMakeDirectory**() function creates a new directory as specified, and takes a short or Long Pathname as a parameter.

C syntax

bool PASCAL LFNMakeDirectory(char szLongName[]);

Parameter

szLongName

The zero terminated Long Pathname directory to create.

Return value

Returns TRUE if the function was successful, otherwise FALSE. Any error will be available through **dlCheckDOSError**().

LFNNextArg()

The **LFNNextArg**() function returns the pointer to the next argument after the current one, and thus can be used to "walk" an argument string. This function treates quoted strings as one argument.

C syntax

char * PASCAL LFNNextArg(char *szArg);

Parameter

szArg

A pointer to the zero terminated argument string.

Return value

Returns the start of the next argument, or the trailing NULL if there are no more arguments.

LFNPresent()

The **LFNPresent()** function checks the kernel to see if it supports Long Filenames. On subsequent calls, it returns the saved result of the first call.

C syntax

bool PASCAL LFNPresent(void)

Parameters

None.

Return value

Returns TRUE if the kernels supports Long Filenames, or FALSE if not.

LFNRemoveDirectory()

The **LFNRemoveDirectory()** function removes the specified directory, and takes a short or Long Pathname as a parameter.

C syntax

bool PASCAL LFNRemoveDirectory(char szLongName[]);

Parameters

szLongName

The zero terminated Long Pathname directory to remove.

Return value

Returns TRUE if the function was successful, otherwise FALSE. Any error will be available through **dlCheckDOSError()**.

LFNRenameFile()

The **LFNRenameFile()** function changes the files name from the old value to the new. This is an alteration of the directory entry only; the actual contents of the file remain unmoved.

C syntax

bool PASCAL LFNRenameFile(char szOldLongName[], char szNewLongName[]);

Parameters

szOldLongName
The zero terminated short or Long Filename file entry to rename.

szNewLongName
The new zero terminated short or Long Filename of the target file.

Return value

Returns TRUE if the function was successful, otherwise FALSE. Any error will be available through **dlCheckDOSError**().

LFNSkipWhite()

The **LFNSkipWhite()** function moves past standard white space characters to the next character that is not white space.

C syntax

char * PASCAL LFNSkipWhite(char *szArg);

Parameter

szArg
A pointer to the zero terminated argument string.

Return value

Returns the character pointer to the next non-whitespace character.

LFNSplitFileName()

The **LFNSplitFileName()** function takes a full path and splits it into the component parts -- the drive, directory, and filename.

C syntax

int PASCAL LFNSplitFileName(char *szSrcName, char *szDrive, char *szDirectory, char *szFileName)

Parameters

szSrcName
The full path of a file. If the drive letter is not included, the current drive will be assumed.

szDrive
A location to store the drive letter and colon (e.g. "D:").

szDirectory
A location to store the full path leading to the file. Relative paths will not be expanded (e.g. "..\").

szFileName

A location to store the file name and extension. Wild cards will not be expanded (e.g. "NAME*.*").

Return value

Returns a byte of flags indicating what actions were performed, and whether any wild cards were found. From DLLFN.H come these defintions:

```
#define SPLIT_WILDCARDS 0x01
#define SPLIT_FILENAME 0x04
#define SPLIT_DIRECTORY 0x08
#define SPLIT_DRIVE  0x10
```

LFNStripArgQuotes()

The **LFNStripArgQuotes()** function removes all matched pairs of quotes from the passed string. Thus, <"hello"" world"> becomes <hello world>. This is a useful function when dealing with quoted parameter strings, but can also be used to process user input.

C syntax

void PASCAL LFNStripArgQuotes(char *pszArg);

Parameters

pszArg

A pointer to the zero terminated argument string.

Return value

None

LFNSubstFunction()

The **LFNSubstFunction()** function performs the three valid subst functions of create, terminate, and query. This function allows for short and Long Filename substs.

C syntax

bool PASCAL LFNSubstFunction(int iFunction, int iDrive, char szLongName[]);

Parameters

iFunction

The function to perform, as defined in DLLFN.H:

```
#define LFN_CREATE_SUBST 0
#define LFN_TERMINATE_SUBST 1
#define LFN_QUERY_SUBST  2
```

iDrive

The DOS drive parameter (0=the current drive, 1=A:, 2=B:, etc.). Note that drive "0" (current drive) is not allowed for function "0" (create subst).

szLongName

The zero terminated Long Pathname directory to subst, or a buffer for the result of the query.

Return value

Returns TRUE if the function was successful, otherwise FALSE. Any error will be available through **dlCheckDOSError()**.

LbaToCHS()

The **LbaToCHS()** function converts from a linear LBA sector to CHS geometry for the specified drive. Note that you must pass the drive information structure, which contains the maximum values, for these calculations to be accurate.

C syntax

bool PASCAL LbaToCHS(PLSIP plsipInfo, ulong ulSector, int *piTrack, uchar *pucHead, uchar *pucSector)

Parameters

plsipInfo

A structure containing the number of sectors on the drive, drive type, and also the standard cylinder, head, and track geometry. See DLINT13.H for a complete description of this structure, which is returned by **DlSmartLbaGetDriveParameters()**.

ulSector

The unsigned long sector to convert.

piTrack

A pointer to the integer where the track number will be returned.

pucHead

A pointer to the unsigned character where the head number will be returned.

pucSector

A pointer to the unsigned character where the sector number will be returned.

Return value

Returns TRUE if the function is successful, or FALSE if any error is encountered.

QuadMultiply()

The **QuadMultiply()** function performs the integer multiplication of two values of up to unsigned long length. The result is returned in a quad word structure.

C syntax

uquad PASCAL QuadMultiply(ulong ulValue1, ulong ulValue2);

Parameters

ulValue1

The multiplicand, which is a number to be multiplied.

ulValue2

The multiplier, which is a number to be multiplied.

Return value

Returns a quad word structure containing the product of the multiplication.

SmartChangeDirectory()

The **SmartChangeDirectory()** function will change to a given directory, and can operate with either standard or Long Filename paths. This function also adjusts for a bug in the current "Windows NT" implementation of the LFN functions, which change the current drive on this call.

C syntax

bool PASCAL SmartChangeDirectory(char * szPathName);

Parameter

szPathName

The zero terminated argument string for the desired path.

Return value

Returns TRUE if the directory exists, otherwise FALSE.

SmartCreateOpenFile()

The **SmartCreateOpenFile()** function is used to create or open a file, and can operate with either standard or Long Filename paths or filenames.

C syntax

bool PASCAL SmartCreateOpenFile(int iModeFlags, int iAttr, int iAction, char szFileName[], int *pHandle, int *pActionTaken);

Parameters

iModeFlags

Various flags for the file open mode, as defined in DLLFN.H:

```
#define OPEN_ACCESS_READONLY             0
#define OPEN_ACCESS_WRITEONLY            1
#define OPEN_ACCESS_READWRITE            2
#define OPEN_ACCESS_RO_NOMODLASTACCESS   4
#define OPEN_SHARE_COMPATIBLE            0
#define OPEN_SHARE_DENYREADWRITE         0x10
#define OPEN_SHARE_DENYWRITE             0x20
#define OPEN_SHARE_DENYREAD              0x30
#define OPEN_SHARE_DENYNONE              0x40
#define OPEN_FLAGS_NOINHERIT             0x80
#define OPEN_FLAGS_NO_BUFFERING          0x100
#define OPEN_FLAGS_NO_COMPRESS           0x200
#define OPEN_FLAGS_ALIAS_HINT            0x400
#define OPEN_FLAGS_NOCRITERR             0x2000
#define OPEN_FLAGS_COMMIT                0x4000
```

iAttr

The desired attribute for the resulting file, as defined in DLLFN.H:

```
#define A_NORMAL          0x00
```

```
#define A_READONLY        0x01
#define A_HIDDEN          0x02
#define A_SYSTEM          0x04
#define A_VOLUME          0x08
#define A_SUBDIR          0x10
#define A_ARCHIVE         0x20
#define A_ALLDIR          0x17
```

iAction

A flag used to indicate whether a Create, Open or Truncate is desired, as defined in DLLFN.H:

```
#define FILE_CREATE       0x10
#define FILE_OPEN         1
#define FILE_TRUNCATE     2
```

szFileName

The zero terminated filename, which may be a standard Short or Long Filename.

pHandle

The returned file handle which can be used to further access the file.

pActionTaken

The returned action taken by the function; whether it Opened, Created or Truncated a file, as defined in DLLFN.H:

```
#define ACTION_OPENED             1
#define ACTION_CREATED_OPENED     2
#define ACTION_REPLACED_OPENED    3
```

Return value

Returns TRUE if the file is opened or created, otherwise FALSE.

SmartDelete()

The **SmartDelete()** function deletes one file, and will use the LFNDeleteFiles() function if possible.

C syntax

bool PASCAL SmartDelete(char * szFileName);

Parameters

szFileName

The zero terminated filename or Long Filename.

Return value

Returns TRUE if the file is deleted, otherwise FALSE.

SmartExpandPath()

The **SmartExpandPath()** function takes the passed short, long, mixed or relative path, and returns the short, long, or mixed version of the path. Depending on the flags, it can also expand the true path of a SUBSTed drive.

C syntax

bool PASCAL SmartExpandPath(int iLFNExpandFlags, char * szPath, char * szExpandedPath);

Parameters

iLFNExpandFlags

If the function finds Long Filenames, it will use LFNGetFullPath(). See that function for a description of these flags.

szPath

The zero terminated source path, with or without Long Filenames and relative components.

szExpandedPath

The location to store the zero terminated result of the function.

Return value

Returns TRUE if the operation is a success, otherwise FALSE.

SmartFindAreAllClosed()

The **SmartFindAreAllClosed()** function checks the status of the Long Filename find handles, and returns TRUE if they are all closed and available.

C syntax

bool PASCAL SmartFindAreAllClosed(void)

Parameters

None

Return value

Returns TRUE only if all find handles are currently available, or closed.

SmartFindClose()

The **SmartFindClose()** function is required to close the handle of a Long FileName find that was created by SmartFindFirst. This is required by most kernel implementations of Long Filenames, which have a limited number of Find handles.

C syntax

bool PASCAL SmartFindClose(PFIND pFindData);

Parameter

pFindData

A structure which contains the find information returned from SmartFindFirst(). See DOSTRUCT.H for a complete definition of this structure.

Return value

Returns TRUE if successful, otherwise FALSE.

SmartFindCloseAll()

The **SmartFindCloseAll()** function finds and closes all Long Filename handles created by SmartFindFirst().

C syntax

void PASCAL SmartFindCloseAll(void);

Parameters

None.

Return value

None.

SmartFindFirst()

The **SmartFindFirst()** function is used to replace the standard C function of findfirst(), and can operate with either standard or LFN file entries and paths.

C syntax

bool PASCAL SmartFindFirst(int nAttrs, char szPathMask[], PFIND pFindData);

Parameters

nAttrs

The DOS file attributes to match, as defined in DLLFN.H:

```
#define A_NORMAL          0x00
#define A_READONLY        0x01
#define A_HIDDEN          0x02
#define A_SYSTEM          0x04
#define A_VOLUME          0x08
#define A_SUBDIR          0x10
#define A_ARCHIVE         0x20
#define A_ALLDIR          0x17
```

szPathMask

The path in which to find the files. May contain a drive letter and colon, and will assume current drive otherwise.

pFindData

A structure to contain the found information. See DOSTRUCT.H for a complete definition of this structure.

Return value

Returns TRUE if any files or directories are found, otherwise FALSE.

SmartFindNext()

The **SmartFindNext()** function is the counterpart to the **SmartFindFirst()** function. It replaces the standard C function of findnext(), and can operate with either standard or LFN file entries and paths.

C syntax

bool PASCAL SmartFindNext(PFIND pFindData)

Parameter

pFindData

A structure which contains the find information returned from **SmartFindFirst().** See DOSTRUCT.H for a complete definition of this structure.

Return value

Returns TRUE if another item is found, otherwise FALSE.

SmartGetCurrentDirectory()

The **SmartGetCurrentDirectory()** function returns the current pathname on the selected drive. The Long Filename path will be used if LFNs are supported in the kernel.

C syntax

bool PASCAL SmartGetCurrentDirectory(int nDrive, char szPathName[]);

Parameters

nDrive

The DOS drive parameter (0=the current drive, 1=A:, 2=B:, etc.).

szPathName

A location to store the current pathname. This should be a large enough space for the maximum path length.

Return value

Returns TRUE if the function is successful, or FALSE otherwise.

SmartGetDriveFreeSpace()

The **SmartGetDriveFreeSpace()** function returns the free space on the disk, along with other values that can be used to translate that value from Clusters to Sectors or Bytes or a percentage of the total disk space available.

C syntax

bool PASCAL SmartGetDriveFreeSpace(int drive, ulong *pulSectorsPerCluster, ulong *pulFreeClusters, ulong *pulBytesPerSector, ulong *pulTotalClusters);

Parameters

drive
The DOS drive parameter (0=the current drive, 1=A:, 2=B:, etc.).

**pulSectorsPerCluster*
A pointer to the unsigned long which will receive the Sectors per Cluster value.

**pulFreeClusters*
A pointer to the unsigned long which will receive the Free Clusters value.

**pulBytesPerSector*
A pointer to the unsigned long which will receive the Bytes per Sector value.

**pulTotalClusters*
A pointer to the unsigned long which will receive the count of all the clusters on the drive.

Return value

Returns FALSE if the function was unable to determine or acces the drive.

SmartGetFileAttributes()

The **SmartGetFileAttributes()** function returns the attributes of a given file, and can operate with either standard or Long Filename paths or filenames.

C syntax

bool PASCAL SmartGetFileAttributes(char szName[], unsigned *pAttributes);

Parameters

szName
The zero terminated filename, which may be a standard Short or Long Filename.

pAttributes
The returned attributes of the file, as defined in DLLFN.H:

```
#define A_NORMAL          0x00
#define A_READONLY        0x01
#define A_HIDDEN          0x02
#define A_SYSTEM          0x04
#define A_VOLUME          0x08
#define A_SUBDIR          0x10
#define A_ARCHIVE         0x20
#define A_ALLDIR          0x17
```

Return value

Returns TRUE if successful, otherwise FALSE.

SmartGetLastAccessDate()

The **SmartGetLastAccessDate()** function returns the last accessed date from the LFN find structure. If this is not an LFN kernel, it will return the only appropriate date, which is that of file creation.

C syntax

unsigned PASCAL SmartGetLastAccessDate(LPFIND lpFindData);

Parameters

lpFindData

A structure which contains the find information returned from **SmartFindFirst()**. See DOSTRUCT.H for a complete definition of this structure.

Return value

Returns an unsigned integer containing the last accessed date.

SmartMakeDirectory()

The **SmartMakeDirectory()** function will create the given directory, and can operate with either standard or Long Filename paths.

C syntax

bool PASCAL SmartMakeDirectory(char szPathName[]);

Parameter

szPathName

The zero terminated argument string for the desired directory path.

Return value

Returns TRUE if the directory was created, otherwise FALSE.

SmartRemoveDirectory()

The **SmartRemoveDirectory()** function will remove the given directory, and can operate with either standard or Long Filename paths.

C syntax

bool PASCAL SmartRemoveDirectory(char szPathName[]);

Parameter

szPathName

The zero terminated argument string for the directory path to be removed.

Return value

Returns TRUE if the directory was successfully removed, otherwise FALSE.

SmartRenameFileOrDirectory()

The **SmartRenameFileOrDirectory()** function changes the name of a file or directory from the old value to the new. This is an alteration of the directory entry only; the actual contents of the file or directory remain unmoved.

C syntax

bool PASCAL SmartRenameFileOrDirectory(char * szOldName, char * szNewName);

Parameters

szOldName
The zero terminated short or Long Filename file entry to rename.

szNewName
The new zero terminated short or Long Filename of the target file.

Return value

Returns TRUE if the file or directory was successfully renamed, otherwise FALSE.

SmartWildcardDelete()

The **SmartWildcardDelete()** function removes any matching short or Long Filenames from the drive and/or path specified in the szFileName.

C syntax

bool PASCAL SmartWildcardDelete(char * szFileName);

Parameters

szFileName
The zero terminated short or Long Filename, optionally including wildcards.

Return value

Returns TRUE if all matching files are successfully removed, otherwise FALSE.

ZeroQuad()

The **ZeroQuad()** function sets all four bytes of the passed quad word to zero.

C syntax

void PASCAL ZeroQuad(uquad * pQuad);

Parameter

pQuad
Pointer to a quad word structure.

Return value

none

TCP/IP Basic API Reference (CAPI)

TCP/IP Basic API Overview

This chapter describes the TCP/IP BASIC API also known as the COMPATIBLE API (CAPI), which is compatible with a wide range of third party TCP/IP applications, and contains descriptions for each of the supported functions. The function descriptions are preceded by introductory information that provides some background on the implementation of the COMPATIBLE API. The definitions and prototypes for the C environment are supplied in CAPI.H and COMPILER.H, while the implementation of the C interface is in CAPI.C and _CAPI.C. The COMPATIBLE API provides an interface to the socket, name resolution, ICMP ping, and kernel facilities provided by the Datalight DOS SOCKETS product.

A *socket* is an end-point for a connection and is defined by the combination of a host address (also known as an IP address), a port number (or communicating process ID), and a transport protocol, such as UDP or TCP.

Two connected SOCKETS using the same transport protocol define a connection. The API uses a socket handle, sometimes referred to as simply a socket. Previously, the socket handle has been referred to as a network descriptor. The socket handle is required by most function calls in order to access a connection. Two types of SOCKETS can be used: 1) a DOS compatible socket, previously referred to as a local network descriptor, which uses a DOS file handle, and 2) a normal socket (previously referred to as a global network descriptor) which does not use a DOS file handle.

New designs should always use normal SOCKETS. A socket handle is obtained by calling the **GetSocket**() function. A socket handle can only be used for a single connection. When no longer required, such as when a connection has been closed, the socket handle must be released by calling **ReleaseSocket**(). DOS compatible socket handles are in the range 0 to 31, although 0 to 4 are normally be used by the C runtime for DOS files like **stdin** and **stdout**. Normal socket handles are positive numbers greater than 63.

Types of Service

SOCKETS can be used with one of two service types:

1. STREAM (using TCP). Refer also to “Using SOCK_STREAM and SOCK_DGRAM Service” on page 99.
2. DATAGRAM (using UDP).

A stream connection provides for the bi-directional, reliable, sequenced, and unduplicated flow of data without record boundaries. No broadcast facilities can be used with a stream connection.

A datagram connection supports bi-directional flow of data that is not guaranteed to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram connection is that record boundaries in data are preserved. Datagram connections closely model the facilities found in many contemporary packet switched networks such as Ethernet. Broadcast messages may be sent and received when using IPv4. Multicast messages may be sent and received when using IPv4 or IPv6.

Establishing Remote Connections

To establish a connection, one side (the server) must execute a **ListenSocket()** and the other side (the client) a **ConnectSocket()**. A connection consists of the local socket / remote socket pair. It is therefore possible to have a connection within a single host as long as the local and remote *port* values differ.

Each host in an IP network must have at least one host address also known as an IP address. When a host has more than one physical connection to an IP network or IPv6 is used, it may have more than one IP address. An IP address must be unique within a network.

An IP address is 32 bits in length for IPv4 and 128 bits for IPv6. The IPAD union defines either an IPv4 address or an IPv6 address in host byte order i.e. the least significant byte is stored first (lowest memory address) and the most significant byte is stored last (highest memory address).

A port number is 16 bits long. A value of zero means "any" while a binary value of all 1s means "all." The latter value is used for broadcasting purposes.

Using the NET_ADDR structure conveys the addresses (host/port) to be used in a connection. The local host is not specified; it is implied. If a value of 0 is specified for the IP address, any remote IP address is accepted; and if a value of 0 is specified for a remote port, any remote port is accepted. This is normally the case when a server is listening for an incoming call. If a value of 0 is specified for *wLocalPort* in the case of a client calling **ConnectSocket()**, a unique port number is assigned by the TCP/IP stack. For IPv4 only operation *dwRemoteHost* contains the IP address of the remote host in network byte order (most significant byte first, least significant byte last). This is compatible with the format used in pre-IPv6 implementation versions of Sockets. For IPv4 and/or IPv6 extended operations *dwRemoteHost* is used to convey the IP address length (4 or 16) and *sIpAddr* is used to convey the IPv4 or IPv6 address in host byte order i.e. the least significant byte first, the most significant byte last. Note that old IPv4 only applications should still work without problem since 4.0.0.0 and 16.0.0.0 are both invalid IPv4 host addresses.

Using STREAM and DATAGRAM Services

When using the STREAM service (TCP), bi-directional data can be sent using the **WriteSocket()** function and received using the **ReadSocket()** function until one side performs an **EofSocket()** after which that side cannot send any more data , but can still receive data until the other side performs an **EofSocket()**, **AbortSocket()** or **ReleaseSocket()**.

When using the DATAGRAM service, datagrams can be sent without first establishing a "connection". In fact UDP provides a "connectionless" service although the connection paradigm is used. In addition to **ReadSocket()** and **WriteSocket()**, **ReadFromSocket()** and **WriteToSocket()** can be used. In this case **EofSocket()** has no meaning and returns an error.

Blocking and Non-blocking Operations

The default behavior of socket functions is to block on an operation and only return when the operation has completed. For example, the **ConnectSocket()** function only returns after the connection has been performed or an error is encountered. This behavior applies to most socket function calls, such as **ReadSocket()** and even **WriteSocket()**, and especially on STREAM connections.

In many, if not most applications, this behavior is unacceptable in the single-threaded DOS environment and must be modified. This modification can be accomplished by either:

1. Specifying the NET_FLG_NON_BLOCKING flag on ReadSocket() and WriteSocket() calls, or
2. Making all operations on a socket non-blocking by calling SetSocketOption() with the NET_OPT_NON_BLOCKING option.

If a non-blocking operation is performed, the function always returns immediately. If the function could not complete without blocking, an error is returned with *iNetErrNo* containing ERR_WOULD_BLOCK. This error should be regarded as a recoverable error and the operation should be retried, preferably at some later time.

Blocking Operations with Timeouts

A possible alternative to using non-blocking operations is to use blocking operations with timeouts. This is done by calling **SetSocketOption()** with the NET_OPT_TIMEOUT option, in which case the function blocks for the specified time, or until completed, whichever occurs first. If the specified timeout occurs first, an error is returned with *iNetErrNo* containing ERR_TIMEOUT and the operation must be retried. Use non-blocking operations rather than timeouts, although they may be somewhat more difficult to implement.

Asynchronous Notifications/Callbacks

Asynchronous notifications or callbacks can be used in cases where the polling implied by non-blocking operation is not desirable, when immediate action is required, when a network operation completes, or when a SOCKETS application runs as a TSR. However, such notifications may be difficult to use and the programmer must be careful to avoid system crashes resulting from improper use.

The **SetAsyncNotification()** function sets functions to be called on specific events, such as opening and closing of STREAM connections and receiving data on STREAM and DATAGRAM connections. The **SetAlarm()** function is called to set a function to be called when a timer expires. Asynchronous notifications are disabled by the **DisableAsyncNotification()** function and enabled by the **EnableAsyncNotification()** function. For more details on the operation and pitfalls associated with callbacks, refer to the description of **SetAsyncNotification()**.

ResolveName(), **GetDCSocket()**, **ConvertDCSocket()**, **ReleaseSocket()** on a DC socket, **ConnectSocket()** with a socket value of –1, **ListenSocket()** with a socket value of –1 and **GetAddressInfo()** with the AI_HOSTTAB flag set, all call DOS. For this reason, these functions should not be called from within a callback or an interrupt service routine.

IP Address Resolution

Three functions are provided for IP address resolution.

ParseAddress() converts a dotted decimal address to a 32-bit IP address.

ResolveName() converts a symbolic host name to a 32-bit IP address using a host table lookup. If that fails and a domain server is configured, a DNS lookup is performed. **ResolveName()** calls DOS to perform a host table lookup and blocks while doing a DNS lookup.

GetAddressInfo() converts a symbolic host name to either a 32-bit IPv4 address or a 128-bit IPv6 address depending on the name and the specific kernel used. The method(s) used to resolve the name can be specified. If host table lookup is specified, **GetAddressInfo()** calls DOS to perform the lookup, If DNS lookup is specified, **GetAddressInfo()** blocks while doing the lookup.

Obtaining SOCKETS Kernel Information

You can obtain information on the SOCKETS TCP/IP kernel by the **GetKernelInformation()**, **GetVersion()** and **GetKernelConfig()** functions. You can unload the kernel by **ShutDownNet()**.

Error Reporting

In general, the C functions implementing the compatible API return a value of –1 if the return type is **int** and an error is encountered, in which case, the actual error code is returned in a common variable *iNetErrNo*. In some cases, *iSubNetErrNo* is also used.

Any API call may fail with an error code of ERR_API_NOT_LOADED or ERR_RE_ENTRY. ERR_RE_ENTRY is returned when the SOCKETS kernel has been interrupted. This condition can occur only when the API is called from an interrupt service routine. Programs designed for this type of operation, such as TSR programs activated by a real time clock interrupt, should be coded to handle this error by re-trying the function at a later stage.

Low Level Interface to the Compatible API

Low level functions to access the Compatible API may be used. In this case, the compatible API is called by setting up the CPU registers and executing a software interrupt. The default interrupt is 61 hexadecimal, but may be relocated when SOCKETS is loaded. If the actual interrupt is not known, a search may be performed for it. Refer to the source file CAPI.C for more details.

On entry, AH contains a number specifying the function to perform. On return, the carry flag is cleared on success and set on failure.

Alternatives to the Compatible API

Additional programming interfaces are available for use with SOCKETS. The first is an earlier revision of CAPI, now called CAPIOLD. This interface is provided to maintain compatibility with applications developed for SOCKETS 1.0. It is superseded by CAPI, which is better-documented and easier to use. Both CAPI and CAPIOLD rely on an internal array of socket descriptors, which must be configured at compile-time. This can use excess memory if your application rarely uses a large number of SOCKETS simultaneously. In addition, it is advised that these APIs do not deal well with mixing both blocking and non-blocking SOCKETS in one application.

The second interface is an even more basic API called the Proprietary API. It is a more natural kernel interface, which hides fewer details from the programmer. As a result, it is more difficult to work with, and should be used only when its extended features and lowered memory footprint are required. The documentation is only provided inside the API.H source file.

The most advanced API is the TCP/IP SOCKETS API which is an implementation of a subset of the BSD Sockets API as well as the Winsock API. The SOCKETS API is implemented as a layer on top of CAPI and thus uses more memory, but in return it provides a well-known API. See the

section “TCP/IP Advanced API Reference (BSD TCP/IP Sockets)” on page 98 for a complete description.

The industry-standard NETBIOS API is also available.

Porting for Compilers

Compiler specific functions have been written into the compiler.h. Modifications for compilers other than the supplied Borland BC5.2 compiler and any listed within compiler.h need to happen within this file. Datalight will offer any assistance we can to help with porting to other compilers but our expertise exists within the supplied Borland compiler.

DJGPP and DPMI Support

The Compatibility API contains support for the GNU-C compiler, DJGPP and allows 32-bit DOS programs running in Protected Mode to provide TCP/IP communications over Datalight Sockets. While most of the implementation details are internal, there are a few details that need to be explained.

1. Necessary setup
2. Alarm and Asynchronous Callbacks
3. New and Updated Macros

1. Necessary Setup

CAPI Setup is fairly simple. The files "DPMI.H" and "GO32.H" must be included before "COMPILER.H" and "CAPI.H":

```
#include <dpmi.h>
#include <go32.h>
#include "compiler.h"
#include "capi.h"
.
.
<rest of program>
.
.
```

2. Alarm and Asynchronous Callbacks

Alarm and asynchronous callbacks are used by Sockets to notify an application of an event, or pending event. Sockets cannot execute the callback directly, because the callback is running in Protected Mode.

Instead, CAPI allocates a Real Mode stub that Sockets calls. This stub proceeds to switch from Real to Protected Mode and executes the callback. While this portion of the API is internal, the callback itself must be written differently in Protected Mode than it would be written in Real Mode. In Real Mode, the data passed to the callback is in registers and can be accessed directly from those registers. In Protected Mode, the registers are stored in a structure that was designated when the Real Mode stub was allocated. This register structure is accessed by the global variable "cb_regs" and is of the type "__dpmi_regs". Any register values passed to the callback are in this

structure and any return register values should be updated in this structure. The format of the __dpmi_regs structure can be found in the file "DPMI.H".

3. New and Updated Macros

Several new macros have been added and several previous macros have been updated to support DJGPP. These are for reference only. There is no requirement to use these macros, but they are available in COMPILER.H.

The macros that have been updated for DJGPP are:

* D_FAR expands to nothing under DJGPP.

* LOADDS expands to nothing under DJGPP.

* ISR_ROUTINE expands to nothing under DJGPP.

* REAL_MODE_SEGMENT expects a _go32_dpmi_seginfo variable to be passed that can retrieve the computed Real Mode segment of an allocated buffer.

* REAL_MODE_OFFSET expects a _go32_dpmi_seginfo variable to be passed that can retrieve the computed Real Mode offset of an allocated buffer.

* MAKE_REAL_POINTER takes the segment and offset and creates a physical Real Mode pointer (()segment << 4) + offset).

* _ASM defines the assembly language command for DJGPP.

The macros that have been added are:

* __ATTR__

Protected Mode: allows a structure to be created without inserting gaps to align for faster access.

Real Mode: expands to nothing.

*.DPMIINFO(i)

Protected Mode: creates i as a _go32_dpmi_seginfo variable.

Real Mode: creates i as a pointer for use with the REAL_MODE macros.

* DPMIINFO_INIT(i)

Protected Mode: initializes i to zero's.

Real Mode: expands to nothing

* DPMIINFO_ALLOC(i,s)

Protected Mode: allocates i as a Protected Mode transfer buffer of size s. iNetErrNo is set to ERR_NO_MEM if allocate fails.

Real Mode: expands to nothing

* DPMIINFO_REALLOC(i,s)

Protected Mode: resizes i, which is a Protected Mode transfer buffer, to size s. iNetErrNo is set to ERR_NO_MEM if resize fails.

Real Mode: expands to nothing.

* DPMIINFO_SET(i, p, s)

Protected Mode: copy s bytes of data from Protected Mode buffer p to transfer buffer i.

Real Mode: assigns pointer p to i, which was allocated with

DPMIINFO_INIT.

* DPMIINFO_GET(i, p, s)

Protected Mode: copy s bytes of data from transfer buffer i to Protected Mode buffer p.

Real Mode: expands to nothing.

* DPMIINFO_FREE(i)

Protected Mode: releases transfer buffer i from memory.

Real Mode: expands to nothing

* DPMI_CB_INFO(i)

Protected Mode: creates i as a _go32_dpmi_seginfo variable.

Real Mode: defines pointer i for callback info.

* DPMI_CB_ALLOCRETF(i, p, r)

Protected Mode: allocates a callback using _go32_dpmi_seginfo variable i. p is the Protected Mode callback routine and r is the register structure to pass the register values from Sockets.

Real Mode: assigns callback routine p to variable i, which was created with DPMI_CB_INFO.

* DPMI_CB_REGS(r)

Protected Mode: defines r as a __dpmi_regs variable for use with DPMI_CB_ALLOCRETF.

Real Mode: creates r as a pointer to use as a parameter to DPMI_CB_ALLOCRETF.

* GET_TICKS

Protected Mode: uses _farpeekw to get ticks at 0x40:0x6C.

Real Mode: uses direct memory access to get ticks at 0x40:0x6C.

* DPMI_SECTION_EXTERNS

Protected Mode: defines the sections used to lock code and data to prevent swapping when switching from Protected to Real Mode.

Real Mode: expands to nothing.

* DPMI_LOCK_SECTIONS

Protected Mode: locks the sections specified in DPMI_SECTION_EXTERNS.

Real Mode: expands to nothing.

* DebugWritePort(port, val)

This routine does similar things in Real and Protected mode. It writes val to port. This macro is only available when the conditional compiler option DEBUG is defined (-DDEBUG)

Usage Notes

Please refer to the make file provided within the SOCKETS\EXAMPLES directory for command line compiler options.

Function Reference

The following sections describe the individual functions of the Compatible API.

AbortDCSockets

The **AbortDCSockets** function aborts all DOS compatible socket connections.

C syntax

int AbortDCSockets(void);

Return value

Returns 0 on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH ABORT_DC_SOCKETS (0x24).

Low level return parameters

If carry flag is set, AX = error code.

AbortSocket

The **AbortSocket()** function aborts the network connection and releases all resources. This function causes an ungraceful close (reset) on a STREAM connection.

C syntax

int **AbortSocket**(int iSocket);

Parameter

iSocket

Socket handle for the connection.

Return value

Returns socket handle on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH ABORT_SOCKET (0x19).

BX Socket.

Low level return parameters

If carry flag is set, AX = error code.

AcceptSocket

The **AcceptSocket()** function accepts network connections on a listen/accept socket and returns a normal socket on the new connection. The returned socket must be used to operate on the connection. If no incoming connection has been received on a blocking socket, **AcceptSocket()** will block until a connection has been received or a time-out occurs. On a non-blocking socket ERR_WOULD_BLOCK will be returned.

If *psAddr* is non-zero, the address information of the accepted socket is returned in the specified NET_ADDR structure. If the accepted socket represents an IPv6 connection or iType contains the TYPE_EXT flag, the length of the IP address (4 or 16) is returned in psAddr->dwRemoteHost and the IP address in psAddr->sIpAddr in host byte order else the IPv4 address is returned in psAddr->dwRemoteHost in network byte order.

C syntax

int AcceptSocket(int iSocket, int iType, NET_ADDR *psAddr);

Parameter

iSocket

Socket handle for the listen/accept connection.

iType

Type of connection: STREAM

Use STREAM | TYPE_EXT to force extended address operation also for IPv4.

psAddr

Pointer to NET_ADDR structure.

Return value

Returns socket handle on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH ACCEPT_SOCKET (0x66).

BX Socket.

DX Connection mode: STREAM or DataGram.

DS:SI Pointer to NET_ADDR address structure.

Low level return parameters

If carry flag clear, initiated OK, AX = Socket.

If carry flag set, AX = error code.

ConnectSocket

The **ConnectSocket()** function makes a network connection. If *iSocket* is specified as –1, a DOS compatible socket is assigned. In this case only, DOS is called to open a file handle.

If *iSocket* specifies a non-blocking socket or *iType* specifies a DATAGRAM connection, this call returns immediately. In the case of a STREAM connection, the connection may not yet be established. **ReadSocket()** can be used to test for connection establishment. As long as **ReadSocket()** returns an ERR_NOT_ESTAB code, the connection is not established. A good return or an error return with ERR_WOULD_BLOCK indicates an established connection. A more complex method uses **SetAsyncNotify()** with NET_AS_OPEN to test for connection establishment. NET_AS_ERROR should also be set to be notified of a failed open attempt.

C syntax

int **ConnectSocket**(int iSocket, int iType, NET_ADDR *psAddr);

Parameter

iSocket
Socket handle for the connection.

iType
Type of connection: STREAM or DATA_GRAM.

psAddr
Pointer to NET_ADDR structure.

Return value

Returns socket on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH CONNECT_SOCKET (0x13).

BX Socket.

DX Connection mode: STREAM or DATA_GRAM.

DS:SI Pointer to NET_ADDR address structure.

Low level return parameters

If carry flag clear, initiated OK, AX = Socket.

If carry flag set, AX = error code.

ConvertDCSocket

The **ConvertDCSocket()** function changes a DOS compatible socket handle into a normal socket handle. This function calls DOS to close a DOS file handle.

C syntax

int ConvertDCSocket(int iSocket);

Parameter

iSocket

DOS compatible socket handle.

Return value

Returns socket handle on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH CONVERT_DC_SOCKET (0x07).

BX Local socket.

Low level return parameters

AX = Global socket if no error.

DisableAsyncNotification

The **DisableAsyncNotification()** function disables Asynchronous notifications (callbacks).

C syntax

int DisableAsyncNotification(void);

Return value

Returns –1 on error with *iNetErrNo* containing the error. Returns previous state on success, 0 for disabled, 1 for enabled.

Low level calling parameter

AH DISABLE_ASYNC_NOTIFICATION (0x11)

Low level return parameter

AX = previous state, 0 = disabled, 1 = enabled

EnableAsyncNotification

The **EnableAsyncNotification()** function enables asynchronous notifications (callbacks).

C syntax

int EnableAsyncNotification(void);

Return value

Returns –1 on error with *iNetErrNo* containing the error. Returns previous state on success, 0 for disabled, 1 for enabled.

Low level calling parameter

AH ENABLE_ASYNC_NOTIFICATION (0x12)

Low level return parameter

AX = previous state, 0 = disabled, 1 = enabled.

EofSocket

The **EofSocket()** function closes the STREAM (TCP) connection (sends a FIN). After **EofSocket()** has been called, no **WriteSocket()** calls may be made. The socket remains open for reading until the peer closes the connection.

C syntax

int **EofSocket(**int iSocket**)**;

Parameter

iSocket

Socket handle for the connection.

Return value

Returns 0 on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH EOF_SOCKET (0x18)

BX Socket

Low level return parameters

If carry flag is set, AX = error code.

FlushSocket

The **FlushSocket()** function flushes any output data still queued for a TCP connection. This defeats the Nagle heuristic and should be used with care.

C syntax

int **FlushSocket(**int iSocket**)**;

Parameter

iSocket

Socket handle for the connection.

Return value

Returns 0 on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH FLUSH_SOCKET (0x1e)

BX Socket

Low level return parameters

If carry flag set, AX = error code.

GetAddress

The **GetAddress()** function gets the local IP address of a connection. In the case of a single interface host, this is the IP address of the host. In the case of more than one interface, the IP address of the interface being used to route the traffic for the specific connection is given.

C syntax

DWORD GetAddress (int iSocket);

Parameter

iSocket

Socket handle for the connection.

Return value

Returns IP address on success. Returns 0L on error with *iNetErrNo* containing the error.

Low level calling parameters

AH GET_ADDRESS (0x05)

BX Socket

Low level return parameters

AX:DX = IP address of this host. AX:DX = 0:0 on error.

GetAddressEx

The **GetAddressEx()** function gets the local IP address of a connection using either IPv4 or IPv6. Note that the information may change during the lifetime of a connection and is generally only valid once data has been sent on the connection.

C syntax

int **GetAddressEx**(int iSocket, NET_ADDR *pAddr);

Options

iSocket

Socket handle for the connection.

pAddr

Pointer to NET_ADDR structure to receive information.

Return values

Returns 0 and NET_ADDR structure filled in on success. The length of the IP address is returned in pAddr->dwRemoteHost with a value of 4 for IPv4 and 16 for IPv6. The IP address is returned in pAddr->sIpAddr. Note that only 4 address bytes will be returned in the case of IPv4; it is therefore good practice to fill the NET_ADDR structure with zeroes before calling **GetAddressEx()**.

Returns –1 with *iNetErrNo* containing the error on failure.

Low level calling parameters

AH GET_ADDRESS _EX(0x6b).

BX Socket.

DS:DX Pointer to NET_ADDR address structure.

Low level return parameters

If carry flag is clear, the address structure is filled in

If carry flag is set, AX = error code

GetAddressInfo

The **GetAddressInfo()** function resolves a symbolic IP address. All the methods for which the corresponding bit is set in the flags word, will be tried until the name is resolved or the methods exhausted. The order in which the methods will be used is the same as the order in which the methods are described below except when DNS_IPV4, DNS_IPV6 and DNS_IPV6_FIRST are all set, in which case DNS_IPV6 will be tried before DNS_IPV4. Not setting AI_HOSTTAB will remove the constraint that DOS may be called.

C syntax

WORD **GetAddressInfo**(char *pszName,WORD wFlags, NET_ADDR *psAddress);

Options

pszName

Pointer to string containing symbolic name.

wFlags

Flag bits specifying method of resolution*:*

AI_PARSE Parse numeric name (IPv4 dotted decimal, IPv6 hex notation).

AI_HOSTTAB Use host table to resolve (HOSTS file).

DNS_IPV4 Use DNS to resolve IPv4 address (Record type A).

DNS_IPV6 Use DNS to resolve IPv6 address (Record type AAAA).

DNS_IPV6_FIRST Try IPv6 first if both DNS_IPV4 and DNS_IPV6 are set.

psAddress

Pointer to NET_ADDR structure to receive IP address.

Return value

Returns length of IP address (4 or 16) on success, 0 on failure with iNetErrNo containing the error code.

Low level calling parameters

AH GET_ADDRESS_INFO (0x6D).

BX Flags word.

DS:DX Pointer to string containing symbolic name.

ES:DI Pointer to NET_ADDR structure to receive IP address.

Low level return parameters

If carry flag is clear, AX = IP address length.

If carry flag is set, AX = error code.

GetBusyFlag

The **GetBusyFlag** function returns the busy status of SOCKETS. **GetBusyFlag** is callable at a low level only; there is no high-level function.

Low level calling parameters

AX GET_BUSY_FLAG

Low level return parameters

ES:SI Pointer to the busy flag byte.

Examine only the four low-order bits. A non-zero value indicates that SOCKETS is currently busy. A value greater than 1 indicate that SOCKETS is not only busy, but is re-entered.

GetDCSocket

The **GetDCSocket**() function gets a DOS-compatible socket handle. This function calls DOS to open a DOS file handle.

C syntax

int GetDCSocket(void);

Return value

Returns socket handle on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH GET_DC_SOCKET (0x22).

Low level return parameters

If carry flag is clear, AX = Socket.

If carry flag is set, AX = error code.

GetKernelConfig

The **GetKernelConfig()** function gets the kernel configuration.

C syntax

int **GetKernelConfig** (KERNEL_CONFIG *psKc);

Parameter

psKc

Pointer to KERNEL_CONFIG structure.

bKMaxTcp	Number of TCP sockets allowed.
BKMaxUdp	Number of UDP sockets allowed.
bKMaxIp	Number of IP sockets allowed (0).
bKMaxRaw	Number of RAW_NET sockets allowed (0).
bKActTcp	Number of TCP sockets in use.
bKActUdp	Number of UDP sockets in use.
bKActIp	Number of IP sockets in use (0).
bKActRaw	Number of RAW_NET sockets in use (0).
wKActDCS	Number of active Dos Compatible Sockets.
wKActSoc	Number of active normal Sockets.
bKMaxLnh	Maximum header on an attached network.
bKMaxLnt	Maximum trailer on an attached network.
bKLBUF_SIZE	Size of a large packet buffer.
bKNnet	Number of network interfaces attached.
dwKCticks	Milliseconds since kernel started.
dwKBroadcast	IP broadcast address in use.

Return value

Returns 0 on success with KERNEL_CONFIG structure filled in, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH GET_KERNEL_CONFIG (0x2A).

DS:SI pointer to kernel_conf structure.

Return

KERNEL_CONF structure filled in.

GetKernelInformation

The **GetKernelInformation()** function gets specified information from the kernel.

C syntax

int **GetKernelInformation** (int iSocket,BYTE bCode,BYTE bDevID,void *pData,WORD *pwSize);

Options

iSocket
Socket handle for K_INF_TCP_CB; otherwise ignored..

bCode
Code specifying kernel info to retrieve:

K_INF_HOST_TABLE	name of file containing host table.
K_INF_DNS_SERVERS	IP addresses of DNS Servers.
K_INF_TCP_CONS	number of Sockets (DC + normal).
K_INF_BCAST_ADDR	broadcast IP address.
K_INF_IP_ADDR	IP address of first interface.
K_INF_SUBNET_MASK	netmask of first interface.
K_INF_TCP_CB	TCB of STREAM socket (defined in API.H)
K_INF_DOMAIN	default domain string
K_INF_DNS_CMPS	DNS completion list
K_INF_HOSTNAME	host name
K_INF_LOCAL_PORT	next available local port
K_INF_MASTER_TICK	master tick used for timing
K_INF_VARPTR	variable pointer
K_INF_VARBLOCK	variable block
K_INF_MEMBLOCK	memory block
K_INF_SETMEM	set memory block
K_INF_MAC_ADDR	MAC (Ethernet) address
K_INF_REMOTE_IP	remote IP address on PPP

bDevID
Index of interface where applicable, normally 0.

pData
Pointer to data area to receive kernel information.

puSize
Pointer to WORD containing length of data area.

Return values

On success returns 0 with data area and size word filled in. Returns –1 with *iNetErrNo* containing the error on failure.

Low level calling parameters

AH	GET_KERNEL_INFO (0x02)
DS:SI	Pointer to data area to receive kernel information.
ES:DI	Pointer to WORD containing length of data area.
DH	Code specifying kernel info to retrieve.
K_INF_HOST_TABLE	Gets name of file containing host table.

K_INF_DNS_SERVERS	Gets IP addresses of DNS Servers.
K_INF_TCP_CONS	Gets number of Sockets (DC + normal).
K_INF_BCAST_ADDR	Gets broadcast IP address.
K_INF_IP_ADDR	Gets IP address of first interface.
K_INF_SUBNET_MASK	Gets netmask of first interface.
K_INF_TCP_CB	Gets TCB of STREAM socket (defined in API.H)

Low level return parameters

If no error, data area is filled in as well as the size word.

GetNetInfo

The **GetNetInfo()** function gets information about the network.

C syntax

int **GetNetInfo(**int iSocket, NET_INFO *psNI**)**;

Parameter

iSocket

Socket handle for the connection.

psNI

Pointer to NET_INFO structure. The following members of NET_INFO are obtained:

DwIpAddress
dwIpSubnet
iUp
iLanLen
pLanAddr

Return value

Returns 0 with NET_INFO structure filled in on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH GET_NET_INFO (0x06).

DS:SI Pointer to netinfo structure.

Low-level return

netinfo structure filled in.

GetPeerAddress

The **GetPeerAddress()** function gets peer address information on a connected socket.

If *psAddr* is non-zero, the address information of the connected socket is returned in the specified NET_ADDR structure. If the connected socket represents an IPv6 connection, the length of the IP address (16) is returned in psAddr->dwRemoteHost and the IP address in psAddr->sIpAddr in host byte order. For an IPv4 connection the IPv4 address is returned in psAddr->dwRemoteHost in network byte order.

C syntax

int **GetPeerAddress**(int iSocket, NET_ADDR *pAddr);

Options

iSocket

Socket handle for the connection.

pAddr

Pointer to NET_ADDR structure to receive address information.

Return values

Returns 0 and NET_ADDR structure filled in on success. Returns –1 with *iNetErrNo* containing the error on failure.

Low level calling parameters

AH GET_PEER_ADDRESS (0x16).

BX Socket.

DS:DX Pointer to NET_ADDR address structure.

Low level return parameters

If carry flag is clear, the address structure is filled in

If carry flag is set, AX = error code

GetPeerAddressEx

The **GetPeerAddressEx()** function gets peer address information on a connected socket.

If *psAddr* is non-zero, the address information of the connected socket is returned in the specified NET_ADDR structure. The length of the IP address (4 or 16) is returned in psAddr->dwRemoteHost and the IP address in psAddr->sIpAddr in host byte order.

C syntax

int **GetPeerAddressEx**(int iSocket, NET_ADDR *pAddr);

Options

iSocket

Socket handle for the connection.

pAddr

Pointer to NET_ADDR structure to receive address information.

Return values

Returns 0 and NET_ADDR structure filled in on success. Returns –1 with *iNetErrNo* containing the error on failure.

Low level calling parameters

AH GET_PEER_ADDRESS_EX (0x6e).

BX Socket.

DS:DX Pointer to NET_ADDR address structure.

Low level return parameters

If carry flag is clear, the address structure is filled in

If carry flag is set, AX = error code

GetSocket

The **GetSocket()** function gets a socket handle.

C syntax

int GetSocket(void);

Return value

Returns socket handle on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH GET_SOCKET (0x29).

Low level return parameters

If carry flag is clear, AX = Socket.

If carry flag is set, AX = error code.

GetVersion

The **GetVersion()** function gets version number of the Compatible API.

C syntax

int GetVersion(void);

Return value

Returns 0x214 on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH GET_NET_VERSION (0x0F).

Low level return parameters

AX = 0x214

ICMPPing

The **ICMPPing()** function sends an ICMP ping (echo request) to an IPv4 host and waits until a response is received or for six seconds if no response is received. **ICMPPing()** is always a blocking function.

C syntax

int **ICMPPing**(DWORD dwHost, int iLength);

Options

dwHost

IP address of host to ping.

iLength

Number of data bytes in ping request.

Return value

Returns 0 on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH ICMP_PING (0x30).

CX number of data bytes in ping request.

DX:BX IP address of host to ping.

Low level return parameters

If carry flag is set, AX = error code.

ICMPPingEx

The **ICMPPingEx()** function sends an ICMP ping (echo request) to either an IPv4 or IPv6 host passed in the NET_ADDR structure pointed to by psHost and waits until a response is received or for a time specified in wWait if no response is received. **ICMPPingEx()** blocks while waiting for a response. If wWait is set to zero, **ICMPPingEx()** checks for a response once and if not received yet, returns an error with *iNetErrNo* containing ERR_TIMEOUT . In order to again check for a response without sending an echo request, iLength can be set to a negative value.

C syntax

int **ICMPPingEx**(NET_ADDR *psHost, WORD wSequence, WORD wWait,
int iLength);

Options

psHost

Pointer to NET_ADDR structure containing IP address of host to ping.

wSequence

Sequence number to send. Any received non-matching echo replies will be ignored and flushed.

wWait
Time in milliseconds to block while waiting for a response. Can be set to zero for non-blocking operation.

iLength
Number of data bytes in ping request. If iLength < 0, do not send echo request.

Return value

Returns >=0 on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH ICMP_PING _EX(0x6c).

BX sequence number

CX number of data bytes in ping request.

DX time to wait for response

DS:SI Pointer to NET_ADDR structure

Low level return parameters

If carry flag is set, AX = error code.

IfaceIOCTL

The **IfaceIOCTL()** function controls asynchronous interfaces

C-Syntax

Int **IfaceIOCTL**(char *pszName, WORD wFunction);

Parameters

pszName
Pointer to interface name.

wFunction
Function to perform:

IOCTL_CONNECT	Start dial operation
IOCTL_DISCONNECT	Disconnect modem
IOCTL_ENABLEPORT	Enable communications port
IOCTL_DISABLEPORT	Disable communications port
IOCTL_ENABLEDOD	Enable dial-on-demand
IOCTL_DISABLEDOD	Disable dial-on-demand
IOCTL_GETSTATUS	Get modem/connection status

Return Value

Returns –1 on error, >= 0 if OK.

IOCTL_GETSTATUS returns the following bits:

```
#define ST_DTR 0x01 /* Modem Data Terminal Ready */
#define ST_RTS 0x02 /* Request To Send */
#define ST_CTS 0x04 /* Clear To Send */
#define ST_DSR 0x08 /* Data Set Ready */
#define ST_RI 0x10 /* Ring Indicator */
#define ST_DCD 0x20 /* Data Carrier Detect */
```

```
#define ST_CONNECTED0x40 /* Modem is connected */
#define ST_MODEMSTATE 0x700 /* Modem state mask */
#define STM_NONE0x000 /* No modem on port */
#define STM_IDLE0x100 /* Modem is idle */
#define STM_INITIALIZING0x200 /* Modem is initializing */
#define STM_DIALING 0x300 /* Modem is dialing */
#define STM_CONNECTING 0x400 /* Modem is connecting */
#define STM_ANSWERING 0x500 /* Modem is answering */
#define STPPPP_IN 0x800 /* PPP incoming call */
#define STPPP_STATE 0x7000 /* PPP state */
#define STPPP_DEAD 0x0000 /* PPP dead */
#define STPPP_LCP 0x1000 /* PPP LCP state */
#define STPPP_AP0x2000 /* PPP Authentication state */
#define STPPP_READY 0x3000 /* PPP Ready (IPCP state) */
#define STPPP_TERMINATING 0x4000 /* PPP Terminating */
```

IsSocket

The **IsSocket()** function checks a DOS compatible socket for validity.

C syntax

int **IsSocket**(int iSocket);

Parameter

iSocket

DOS Compatible socket handle for the connection.

Return value

Returns 0 on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH IS_SOCKET (0x0D).

BX Local socket.

Low level return parameters

Carry flag clear if valid.

Carry flag set if not valid, AX = error code.

JoinGroup

The **JoinGroup()** function causes SOCKETS to join an IPv4 multicast group. Once the group has been joined, datagrams sent to the multicast group, will be locally looped back. **JoinGroup()** may be called more than once, in which case **LeaveGroup**() must be called an equal number of times for the group to be left. For new designs, it is preferable to use **JoinGroupEx().**

C syntax

int **JoinGroup**(DWORD dwGroupAddress, DWORD dwIPAddress);

Options

dwGroupAddress

The group address on which to receive multicast datagrams.

dwIPAddress

The IP address for the interface to use. The first interface to be specified in SOCKET.CFG is the default interface in the case where dwIPAddress == 0.

Return value

Returns 0 on success, any other integer value contains the error code.

Low level calling parameters

AH JOIN_GROUP (0x60)

DS:SI Pointer to GROUP_ADDR structure, documented in CAPI.H

Low level return parameters

If carry flag is set, AX = error code

JoinGroupEx

The **JoinGroupEx()** function causes SOCKETS to join an IPv4 or Ipv6 multicast group. Once the group has been joined, datagrams sent to the multicast group, will be locally looped back. **JoinGroupEx()** may be called more than once, in which case **LeaveGroupEx**() must be called an equal number of times for the group to be left.

C syntax

int **JoinGroup**(IPAD *psGroupAddress, DWORD dwInterfaceId, WORD wAddrLen);

Options

psGroupAddress

Pointer to the group address on which to receive multicast datagrams/packets.

dwInterfaceId

The ID for the interface to use. The ID is the position of the interface specification in SOCKET.CFG starting from one and incrementing by one for each interface specification. Note that ID==0 and ID==1 both specify the first interface.

wAddrLen

The length of the multicast group address, 4 for IPv4 and 16 for IPv6.

Return value

Returns 0 on success, any other integer value contains the error code.

Low level calling parameters

AH JOIN_GROUP (0x60)

DS:SI Pointer to GROUP_ADDR structure, documented in CAPI.H

Low level return parameters

If carry flag is set, AX = error code

LeaveGroup

The **LeaveGroup()** function causes SOCKETS to leave an IPv4 multicast group. For new designs, it is preferable to use **LeaveGroupEx().**

C syntax

int **LeaveGroup**(DWORD dwGroupAddress, DWORD dwIPAddress);

Options

dwGroupAddress

The group address on which multicast datagrams are being received.

dwIPAddress

The IP address for the interface being used. The first interface to be specified in SOCKET.CFG is the default interface in the case where dwIPAddress == 0.

Return value

Returns 0 on success, any other integer value contains the error code.

Low level calling parameters

AH LEAVE_GROUP (0x61)

DS:SI Pointer to GROUP_ADDR structure, documented in CAPI.H

Low level return parameters

If carry flag is set, AX = error code

LeaveGroupEx

The **LeaveGroupEx()** function causes SOCKETS to leave an IPv4 or IPv6 multicast group.

C syntax

int **LeaveGroupEx**(IPAD *psGroupAddress, DWORD dwInterfaceId, WORD wAddrLen);

Options

psGroupAddress

Pointer to the group address on which multicast datagrams/packets are being received.

dwInterfaceId

The ID for the interface to use. The ID is the position of the interface specification in SOCKET.CFG starting from one and incrementing by one for each interface specification. Note that ID==0 and ID==1 both specify the first interface.

wAddrLen

The length of the multicast group address, 4 for IPv4 and 16 for IPv6.

Return value

Returns 0 on success, any other integer value contains the error code.

Low level calling parameters

AH LEAVE_GROUP (0x61)

DS:SI Pointer to GROUP_ADDR structure, documented in CAPI.H

Low level return parameters

If carry flag is set, AX = error code

ListenAcceptSocket

The ListenAcceptSocket() function listens for network connections. If *iSocket* is specified as -1, a Dos Compatible socket is assigned. In this case only, DOS is called to open a file handle. This call returns immediately. If *iType* specifies a DATAGRAM connection, this call acts exactly the same as a ListenSocket(). If *iType* specifies a STREAM connection, **AcceptSocket()** must be used to accept incoming connections on *iSocket*, which will remain listening for new connections. Up to *iConnections* incoming connections may be received before an **AcceptSocket()** must be issued to prevent further connections to be refused.

Note that specifying an IP address of all zeroes (0.0.0.0 for IPv4 or :: for IPv6) will accept connections to any local IP address. When using a kernel supporting both transport protocols, both IPv4 and IPv6 connections will be accepted.

C syntax

int **ListenAcceptSocket**(int iSocket, int iType, int iConnections, NET_ADDR *psAddr)

Parameter

iSocket
Socket handle for the connection.

iType
Type of connection: STREAM or DATA_GRAM.

iConnections
Number of connections to be queued pending AcceptSocket(). *iConnections* is silently limited to a maximum of 5.

psAddr
Pointer to NET_ADDR structure.

Return value

Returns socket handle on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH LISTEN_ACCEPT_SOCKET (0x65).

BX Socket.

CX Number of connections.

DX Connection mode: STREAM or DataGram.

DS:SI Pointer to NET_ADDR address structure.

Low level return parameters

If carry flag clear, initiated OK.

If carry flag set, AX = error code.

ListenSocket

The **ListenSocket()** function listens for a network connection. If *iSocket* is specified as –1, a DOS compatible socket is assigned. In this case only, DOS is called to open a file handle.

If *iSocket* specifies a non-blocking socket or *iType* specifies a DATAGRAM connection, this call returns immediately. In the case of a STREAM connection, the connection may not be established yet. **ReadSocket()** can be used to test for connection establishment.

As long as **ReadSocket()** returns an ERR_NOT_ESTAB code, the connection is not established. A good return or an error return with ERR_WOULD_BLOCK indicates connection establishment. A more complex method is to use **SetAsyncNotify()** with NET_AS_OPEN to test for connection establishment. NET_AS_ERROR should also be set to be notified of a failed open attempt.

C syntax

int **ListenSocket(**int iSocket, int iType, NET_ADDR *psAddr);

Parameter

iSocket

Socket handle for the connection.

iType

Type of connection: STREAM or DataGram.

psAddr

Pointer to NET_ADDR structure.

Return value

Returns socket handle on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH LISTEN_SOCKET (0x23).

BX Socket.

DX Connection mode: STREAM or DataGram.

DS:SI Pointer to NET_ADDR address structure.

Low level return parameters

If carry flag clear, initiated OK, AX = Socket.

If carry flag set, AX = error code.

ParseAddress

The **ParseAddress()** function gets an IP address from dotted decimal addresses.

C syntax

DWORD **ParseAddress**(char *pszName);

Parameter

pszName

Pointer to string containing dotted decimal address.

Return value

Returns IP address on success, 0 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH PARSE_ADDRESS (0x50).

DS:DX Pointer to dotted decimal string.

Low level return parameters

AX:DX = IP address.

ReadFromSocket

The **ReadFromSocket()** function reads from the network using a socket and is only intended to be used on DataGram sockets. All datagrams from the IP address and port matching the values in the NET_ADDR structure are returned while others are discarded. A zero value for *dwRemoteHost* is used as a wildcard to receive from any host and a zero value for *wRemotePort* is used as a wildcard to receive from any port. The local port, *wLocalPort* , can not be specified as zero.

In other respects **ReadFromSocket()** behaves the same as **ReadSocket()**.

C syntax

int **ReadFromSocket**(int iSocket, char *pcBuf, WORD wLen, NET_ADDR *psFrom, WORD wFlags);

Options

iSocket

Socket for the connection.

pcBuf

Pointer to buffer to receive data.

wLen

Length of buffer, i.e. maximum number of bytes to read.

psFrom

Pointer to NET_ADDR structure to receive address information about local and remote ports and remote IP address.

wFlags

Flags governing operation. Any combination of:

NET_FLG_PEEK Don't dequeue data.

NET_FLG_NON_BLOCKING Don't block.

NET_FLG_EXT Return extended address information.

Return value

Returns number of bytes read on success, -1 on failure with *iNetErrNo* containing the error code. Note the following anomaly:

If blocking is disabled, a failure with an error code of ErrWouldBlock is completely normal and only means that no data is currently available.

Low level calling parameters

AH READ_FROM_SOCKET (0x1d).

BX Socket.

CX Maximum number of bytes to read.

DX Flags - any combination of.

NET_FLG_PEEK:- Don't dequeue data.

NET_FLG_NON_BLOCKING:- Don't block.

NET_FLG_EXT:- Return extended address information.

DS:SI Pointer to buffer to read into.

ES:DI Pointer to NET_ADDR address structure.

Low level return parameters

If carry flag is clear, AX = CX = number of bytes read.

If carry flag is set, AX = error code.

ReadSocket

The **ReadSocket()** function reads from the network using a socket. **ReadSocket()** returns as soon as any non-zero amount of data is available, regardless of the blocking state. If the operation is non-blocking, either by having used **SetSocketOption()** with the NET_OPT_NON_BLOCKING option or specifying *wFlags* with NET_FLG_NON_BLOCKING, **ReadSocket()** returns immediately with the count of available data or an error of ERR_WOULD_BLOCK.

With a STREAM (TCP) socket, record boundaries do not exist and any amount of data can be read at any time regardless of the way it was sent by the peer. No data is truncated or lost even if more data than the buffer size is available. What is not returned on one call, is returned on subsequent calls. If a NULL buffer is specified or both the NET_FLG_PEEK and NET_FLG_NON_BLOCKING flags are specified, the number of bytes on the receive queue is returned.

In the case of a DataGram (UDP) socket, the entire datagram is returned in one call, unless the buffer is too small in which case the data is truncated, thereby preserving record boundaries. Truncated data is lost. If data is available and both the NET_FLG_PEEK and NET_FLG_NON_BLOCKING flags are specified, the number of datagrams on the receive queue is returned. If data is available and NET_FLG_PEEK is set and a NULL buffer is specified, the number of bytes in the next datagram is returned.

If *psFrom* is non-zero, the address information of the peer is returned in the specified NET_ADDR structure. If the accepted socket represents an IPv6 connection or wFlags contains the NET_FLG_EXT flag, the length of the IP address (4 or 16) is returned in

psAddr->dwRemoteHost and the IP address in psAddr->sIpAddr in host byte order else the IPv4 address is returned in psAddr->dwRemoteHost in network byte order.

C syntax

int **ReadSocket**(int iSocket, char *pcBuf, WORD wLen, NET_ADDR *psFrom, WORD wFlags);

Options

iSocket
Socket handle for the connection.

pcBuf
Pointer to buffer to receive data.

wLen
Length of buffer, i.e. maximum number of bytes to read.

psFrom
Pointer to NET_ADDR structure to receive address information about local and remote ports and remote IP address.

wFlags
Flags governing operation. Any combination of:

NET_FLG_PEEK	Don't dequeue data.
NET_FLG_NON_BLOCKING	Don't block.
NET_FLG_EXT	Return extended address information.

Return value

Returns number of bytes read on success, -1 on failure with *iNetErrNo* containing the error code.

Note: If blocking is disabled, a failure with an error code of ERR_WOULD_BLOCK is completely normal and only means that no data is currently available.

Low level calling parameters

AH READ_SOCKET (0x1b).

BX Socket.

CX Maximum number of bytes to read.

DX Flags - any combination of.

NET_FLG_PEEK:	Don't dequeue data.
NET_FLG_NON_BLOCKING:	Don't block.
NET_FLG_EXT:	Return extended address information.

DS:SI Pointer to buffer to read into.

ES:DI Pointer to NET_ADDR address structure.

Low level return parameters

If carry flag is clear, AX = CX = number of bytes read.

If carry flag is set, AX = error code.

ReleaseDCSockets

The **ReleaseDCSockets** function closes all connections and releases all resources associated with DOS compatible sockets.

C syntax

int ReleaseDCSockets(void);

Return value

Returns 0 on success, -1 on failure with *iNetErrNo* containing the error code.

C syntax

int ReleaseDCSockets(void);

Low level calling parameters

AH RELEASE_DC_SOCKETS (0x09).

Low level return parameters

AX = error code if carry flag is set.

ReleaseSocket

The **ReleaseSocket()** function closes the connection and releases all resources. On a STREAM (TCP) connection, this function should only be called after the connection has been closed from both sides otherwise a reset (ungraceful close) can result.

C syntax

int **ReleaseSocket(**int iSocket**)**;

Parameter

iSocket

Socket handle for the connection.

Return value

Returns socket handle on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH RELEASE_SOCKET (0x08).

BX Socket.

Low level return parameters

AX = error code if carry flag is set

ResolveName

The **ResolveName()** function resolves an IP address from a symbolic name. For new applications **GetAddressInfo()** should be used.

C syntax

DWORD **ResolveName(**char *pszName, char *pcCname, int iCnameLen**);**

Options

pszName

Pointer to string containing symbolic name.

pcCname

Pointer to buffer to receive canonical name.

ICnameLen

Length of buffer pointed to by *pcName.*

Return value

Returns IP address on success, 0 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH RESOLVE_NAME (0x54).

CX Size of buffer to receive canonical name.

DS:DX Pointer to string containing symbolic name.

ES:DI Pointer to buffer to receive canonical name.

Low level return parameters

If carry flag is clear, AX:DX = IP address.

If carry flag is set, AX = error code.

SelectSocket

The **SelectSocket()** function tests all DOS compatible sockets for data availability and readiness to write. A 32-bit DWORD representing 32 DC sockets is filled in for each socket with receive data, and another 32-bit DWORD for DC sockets ready for writing. The least-significant bit represents the socket with value 0 and the most-significant bit represents the socket with value 31. Bits representing unused sockets are left unchanged.

C syntax

int **SelectSocket(**int iMaxid, long *plIflags, long *plOflags**);**

Options

iMaxid

Number of sockets to test.

plIflags

Pointer to input flags indicating receive data availability.

plOflags

Pointer to output flags indicating readiness to write.

Return value

Returns 0 on success with *plIflags and *plOflags filled in with current status, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH SELECT_SOCKET (0x0e).

BX Number of sockets to test.

DS:DX Pointer to DWORD for data availability.

ES:DI Pointer to DWORD for readiness to write.

Low level return parameters

Both DWORDs updated with current status.

SetAlarm

The **SetAlarm()** function sets an alarm timer.

C syntax

int **SetAlarm**(int iSocket, DWORD dwTime, int (far *lpHandler)(), DWORD dwHint);

Options

iSocket
Socket handle for the connection.

dwTime
Timer delay in milliseconds.

lpHandler
Far address of alarm callback. See the description of **SetAsyncNotification**() for the format of the callback function.

dwHint
Argument to be passed to callback function.

Return value

Returns socket handle on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH SET_ALARM (0x2bB.

BX Socket.

CX:DX Timer delay in milliseconds.

DS:SI Address of alarm callback.

ES:DI Argument to be passed to callback.

Low level return parameters

If carry flag is set, AX = error code

See the description of SET_ASYNC_NOTIFICATION for the callback function.

SetAsyncNotification

The **SetAsyncNotification()** function sets an asynchronous notification (callback) for a specific event.

C syntax

int far ***SetAsyncNotification(**int iSocket, int iEvent, int (far *lpHandler)(),DWORD dwHint**);**

Parameter

iSocket

Socket handle for the connection.

iEvent

Event which is being set:

NET_AS_OPEN	Connection has opened.
NET_AS_RCV	Data has been received.
NET_AS_XMT	Ready to transmit.
NET_AS_FCLOSE	Peer has closed connection.
NET_AS_CLOSE	Connection has been closed.
NET_AS_ERROR	Connection has been reset.

lpHandler

Far address of callback function.

dwHint

Argument to be passed to callback function

The handler is not compatible with C calling conventions but is called by a far call with the following parameters:

BX = Socket handle.

CX = Event. If extended address information must be reported for IPv4, NET_AS_EXTENDED should be added to the event value.

ES:DI = dwHint argument passed to SetAsyncNotification() or SetAlarm().

DS:DX = SI:DX = variable argument depending on event:

NET_AS_OPEN NET_AS_CLOSE	Pointer to NET_ADDR address structure.
NET_AS_FCLOSE NET_AS_RCV NET_AS_ALARM	Zero.
NET_AS_XMT	Byte count which can be sent without blocking.
NET_AS_ERROR	Error code –ERR_TERMINATING, ERR_TIME_OUT or ERR_RESET.

Other CAPI functions may be called in the callback, with the exception of ResolveName() which may call DOS. The callback is not compatible with C argument-passing conventions and some care must be taken. Some CPU register manipulation is

required. This can be done by referencing CPU registers, such as _BX, or by means of assembler instructions.

In the callback, the stack is supplied by SOCKETS and may be quite small depending on the /s= command line option when loading SOCKETS. The stack segment is obviously not equal to the data segment, which can cause problems when the Tiny, Small or Medium memory model is used. The simplest way to overcome the problem is to use the Compact, Large or Huge memory model. Other options - use the DS != SS compiler option or do a stack switch to a data segment stack .

If the callback is written in C or C++, the _loads modifier can be used to set the data segment to that of the module, which destroys the DS used for the variable argument. (This is why DS == SI on entry for SOCKETS version 1.04 and later.) An alternate method is to use the argument passed to SetAsyncNotification() in ES:DI as a pointer to a structure that is accessible from both the main code and the callback. If DS is not set to the data segment of the module, then the functions in CAPI.C do not work: Don't use them in the callback.

The callback will probably be performed at interrupt time with no guarantee of reentry to DOS. Do not use any function, such as putchar() or printf(), in the callback which may cause DOS to be called.

It is good programming practice to do as little as possible in the callback. The setting of event flags that trigger an operation at a more stable time is recommended.

Callback functions do not nest. The callback function is not called while a callback is still in progress, even if other CAPI functions are called.

To alleviate the problems in items 2, 3 and 4 above, a handler is provided in CAPI.C that uses the dwHint parameter to pass the address of a C-compatible handler, with a stack that is also C-compatible. This handler is named AsyncNotificationHandler. A user handler named MyHandler below, is called in the normal way with a stack of 500 bytes long. Changing the HANDLER_STACK_SIZE constant in CAPI.C can set the stack size value.

int far **MyHandler(**int iSocket, int iEvent, DWORD dwArg);

SetAsyncNotification(iSocket, iEvent, AsyncNotificationHandler, (DWORD)MyHandler);

Return value

Returns pointer to the previous callback handler on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH SET_ASYNC_NOTIFICATION (0x1F).

BX Socket.

CX Event:

NET_AS_OPEN	Connection has opened.
NET_AS_RCV	Data has been received.
NET_AS_XMT	Ready to transmit.
NET_AS_FCLOSE	Peer has closed connection.
NET_AS_CLOSE	Connection has been closed.
NET_AS_ERROR	Connection has been reset.

DS:DX Address of handler.

ES:DI Argument passed to handler.

Low level return parameters

If carry flag is set, AX = error code, else address of previous handler is returned in ES:DX.

SetSocketOption

The **SetSocketOption()** function sets an option on the socket.

C syntax

int **SetSocketOption(**int iSocket, int iLevel, int iOption, DWORD dwOptionValue, int iLen);

Options

iSocket

Socket handle for the connection.

iLevel

Level of option. This value is ignored.

iOption

Option to set.

NET_OPT_NON_BLOCKING	Set blocking off if dwOptionValue is non-zero.
NET_OPT_TIMEOUT	Set the timeout to dwOptionValue milliseconds. Turn off timeout if dwOptionValue is zero.
NET_OPT_WAIT_FLUSH	Wait for flush if dwOptionValue is non-zero.
NET_OPT_KEEPALIVE	let a connected socket send out a KEEP_ALIVE segment every 30 seconds of no traffic. This tests connectivity and will let a connection time out and be broken using the normal re-try strategy. If using BSD sockets, please use: setsockopt(..,SO_KEEPALIVE,..)

dwOptionValue

Option value.

iLen

Length of *dwOptionValue*, 4 in all cases.

Return value

Returns global socket on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH SET_OPTION (0x20).

BX Socket.

DS:DX Value of option.

DI Option:

NET_OPT_NON_BLOCKING	Set blocking off if dwOptionValue is non-zero.

NET_OPT_TIMEOUT	Set the timeout to dwOptionValue milliseconds. Turn off timeout if dwOptionValue is zero.
NET_OPT_WAIT_FLUSH	Wait for flush if dwOptionValue is non-zero.

Low level return parameters

If carry flag is set, AX = error code.

ShutDownNet

The **ShutDownNet()** function shuts down the network and unloads the SOCKETS TCP/IP kernel.

C syntax

int ShutDownNet(void);

Return value

Returns 0 on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH SHUT_DOWN_NET (0x10).

Low level return parameters

None.

WriteSocket

The **WriteSocket()** function writes to the network using a socket.

C syntax

int **WriteSocket**(int iSocket, char *pcBuf, WORD wLen, WORD wFlags);

Parameter

iSocket

Socket handle for the connection.

pcBuf

Pointer to buffer to containing send data.

wLen

Length of buffer, i.e. number of bytes to write.

wFlags

Flags governing operation; can be any combination of:

NET_FLG_OOB	Send out of band data (TCP only).
NET_FLG_PUSH	Send data even if NET_OPT_WAIT_FLUSH is set. Does not override Nagle heuristic (TCP only).
NET_FLG_NON_BLOCKING	Don't block.
NET_FLG_BROADCAST	Broadcast data (UDP only).

NET_FLG_MC_NOECHO Suppress the local echo of a multicast datagram.

Return value

Returns number of bytes written on success, -1 on failure with *iNetErrNo* containing the error code. The number of bytes actually written on a non-blocking write, can be less than wLen. In such a case, the writing of the unwritten bytes must be retried, preferably after some delay.

Low level calling parameters

AH WRITE_SOCKET (0x1a).

BX Socket.

CX Byte count.

DX Flags - any combination of the following:

NET_FLG_OOB	Send out of band data (TCP only).
NET_FLG_PUSH	Disregard Nagle heuristic (TCP only).
NET_FLG_NON_BLOCKING	Don't block.
NET_FLG_BROADCAST	Broadcast data (UDP only).

DS:SI Pointer to buffer to write.

Low level return parameters

If carry flag is clear, AX = number of bytes sent.

If carry flag is set, AX = error code.

WriteToSocket

The **WriteToSocket()** function writes to the network using a network address (UDP only).

C syntax

int **WriteToSocket**(int iSocket, char *pcBuf, WORD wLen, NET_ADDR *psTo, WORD wFlags);

Options

iSocket

Socket handle for the connection.

pcBuf

Pointer to buffer containing send data.

wLen

Length of buffer, i.e. number of bytes to write.

psTo

Pointer to NET_ADDR structure containing local port to write from and remote port and IP address to write to.

wFlags

Flags governing operation. Any combination of:

NET_FLG_NON_BLOCKING Don't block.

NET_FLG_BROADCAST Broadcast data (UDP only).

Return value

Returns number of bytes written on success, -1 on failure with *iNetErrNo* containing the error code.

Low level calling parameters

AH WRITE_TO_SOCKET (0x1C).

BX Socket.

CX Byte count.

DX Flags:

NET_FLG_NON_BLOCKING Don't block

NET_FLG_BROADCAST Broadcast data (UDP only).

DS:SI Pointer to buffer to write.

ES:DI Pointer to NET_ADDR address structure.

Low level return parameters

If carry flag is clear, AX = number of bytes sent.

If carry flag is set, AX = error code.

Error Codes

Error Value	Error Code	Meaning
NO_ERR	0	No error
ERR_IN_USE	1	A connection already exists
ERR_DOS	2	A DOS error occurred
ERR_NO_MEM	3	No memory to perform function
ERR_NOT_NET_CON	4	Connection does not exist
ERR_ILLEGAL_OP	5	Protocol or mode not supported
ERR_NO_HOST	7	No host address specified
ERR_TIMEOUT	13	The function timed out
ERR_HOST_UNKNOWN	14	Unknown host has been specified
ERR_BAD_ARG	18	Bad arguments
ERR_EOF	19	The connection has been closed by peer
ERR_RESET	20	The connection has been reset by peer
ERR_WOULD_BLOCK	21	Operation would block
ERR_UNBOUND	22	The descriptor has not been assigned
ERR_NO_SOCKET	23	No socket is available
ERR_BAD_SYS_CALL	24	Bad parameter in call
ERR_NOT_ESTAB	26	The connection has not been established
ERR_RE_ENTRY	27	The kernel is in use, try again later
ERR_TERMINATING	29	Kernel unloading

ERR_API_NOT_LOADED	50	SOCKETS kernel is not loaded

TCP/IP Advanced API Reference (BSD TCP/IP Sockets)

TCP/IP SOCKETS API Overview

This chapter describes the SOCKETS API, which is compatible with the BSD Sockets API and also the Winsock API. The definitions and prototypes for the C environment are supplied in SOCKET.H, while the implementation of the C interface is in SOCKET.C. The SOCKETS API is implemented as a layer on top of the Compatible API (CAPI) and provides an interface to the socket and name resolution facilities provided by the Datalight DOS SOCKETS product. It also provides the database functions of BSD Sockets and Winsock.

A *socket* is an end-point for a connection and is defined by the combination of a host address (also known as an IP address), a port number (or communicating process ID), and a transport protocol, such as UDP or TCP.

Two connected SOCKETS using the same transport protocol define a connection. The API uses a socket handle, sometimes referred to as simply a socket. The socket handle is required by most function calls in order to access a connection. The socket handle used is the same as a normal socket as used in CAPI.

A socket handle is obtained by calling the **socket**() function. A socket handle can only be used for a single connection. When no longer required, such as when a connection has been closed, the socket handle must be released by calling **closesocket**(). Socket handles are positive numbers greater than 63.

Types of Service

SOCKETS can be used with one of two service types:

- SOCK_STREAM (using TCP).
- SOCK_DGRAM (using UDP).

A stream connection provides for the bi-directional, reliable, sequenced, and unduplicated flow of data without record boundaries. No broadcast facilities can be used with a stream connection.

A datagram connection supports bi-directional flow of data that is not guaranteed to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram connection is that record boundaries in data are preserved. Datagram connections closely model the facilities found in many contemporary packet switched networks such as Ethernet. Broadcast messages may be sent and received.

Establishing Remote Connections

To establish a connection, one side (the server) must execute a **listen()** and and subsequent **accept()** and the other side (the client) a **connect()**. A connection consists of the local socket /

remote socket pair. It is therefore possible to have a connection within a single host as long as the local and remote *port* values differ.

Each host in an IP network must have at least one host address also known as an IP address. When a host has more than one physical connection to an IP network, it may have more than one IP address. An IP address must be unique within a network.

An IP address is 32 bits in length for IPv4 and 128 bits for IPv6, a port number 16 bits. A value of zero means "any" while a binary value of all 1s means "all." The latter value is used for broadcasting purposes.

Using the **sockaddr** structure conveys the addresses (host/port) to be used in a connection. A local association is performed by the **bind()** function.

Using SOCK_STREAM and SOCK_DGRAM Services

When using the SOCK_STREAM service (TCP), bi-directional data can be sent using the **send()** or **sendto()** functions and received using the **recv()** or **recvfrom()** functions until one side performs a **shutdown(1)** or **shutdown(2)** after which that side cannot send any more data , but can still receive data until the other side performs a **shutdown(1), shutdown(2)** or **closesocket()**.

When using the SOCK_DGRAM service, datagrams can be sent without first establishing a "connection". In fact UDP provides a "connectionless" service although the connection paradigm is used.

Blocking and Non-blocking Operations

The default behavior of socket functions is to block on an operation and only return when the operation has completed. For example, the **connect()** function only returns after the connection has been performed or an error is encountered. This behavior applies to most socket function calls, such as **recv()** and even **send()**, and especially on SOCK_STREAM connections.

In many, if not most applications, this behavior is unacceptable in the single-threaded DOS environment and must be modified. This modification can be accomplished by making all operations on a socket non-blocking by calling **ioctlsocket()** with the FIONBIO option.

If a non-blocking operation is performed, the function always returns immediately. If the function could not complete without blocking, an error is returned with *errno* containing EWOULDBLOCK. This error should be regarded as a recoverable error and the operation should be retried, preferably at some later time.

Out of band data

TCP "out of band" or urgent data is not implemented. Setting the MSG_OOB flag has no effect in **recv(), recvfrom(), send()** or **sendto()**; it will simply be ignored. The SO_OOBINLINE option will also be ignored and **ioctlsocket()** with the SIOCATMARK command, will always return an argument value of 1.

Error Reporting

In general, the C functions implementing the SOCKETS API return a value of SOCKET_ERROR if the return type is **int** and an error is encountered, in which case, the actual error code is returned in a common variable *errno*. ERR_RE_ENTRY is returned when the SOCKETS kernel has been interrupted. This condition can occur only when the API is called from an interrupt service routine. Programs designed for this type of operation, such as TSR programs activated by a real time clock interrupt, should be coded to handle this error by re-trying the function at a later stage.

Other sources of Information

Many good books have been written on the Sockets API. Here are a few:

Pocket Guide to TCP/IP Sockets (C Version)
by Michael J. Donahoo, Kenneth L. Calvert

Windows Sockets Network Programming (Addison-Wesley Advanced Windows Series)
by Bob Quinn, et al; Hardcover

Internetworking with TCP/IP Vol. III Client-Server Programming and Applications-Windows Sockets Version
by Douglas E. Comer, David L. Stevens (Contributor) ;Hardcover.

The Winsock 1.1 help file (WINSOCK.HLP) is also a very useful source of information.

Porting Issues

When porting an application from another BSD Sockets environment like Unix, Linux or Windows (Winsock), a number of issues must be kept in mind. The most important one is that ROM-DOS is a single-user, single-task, single-thread operating system. The use of blocking calls will suspend the system until completion, which may imply an indefinite time under abnormal or even normal conditions. In addition no completion event such as a WSAAsyncSelect windows message for Winsock or a Signal for Unix/Linux is available. Only applications either using non-blocking operations or the **select()** function may be ported successfully. Other applications must be adapted to follow these guidelines.

Unlike Winsock and like BSD Sockets, an error number is returned in the *errno* variable and is only valid directly after an API call. When writing portable code to run on both SAPI and Winsock, a simple #define can normally be used i.e.

```
#ifdef _Windows
#define Errno WSAGetLastError()
#else
#define Errno errno
#endif
.
.
if (Errno == WSAEWOULDBLOCK)
{
        .
        .
```

```
}
.
.
```

Like in Winsock both the WSAE... of Winsock and the E... error definitions of BSD may be used e.g. WSAEWOULDBLOCK and EWOULDBLOCK. The actual error numbers are the same as that of Winsock, except in cases of DOS error code conflicts e.g. WSAEINVAL has the same value as the DOS EINVAL. Always using the symbolic value and not numeric values, will avoid potential problems.

The function gethostbyaddr() will always fail with errno == WSANO_DATA.

All the file/socket operations of BSD Sockets must be translated to the *socket() versions as used in Winsock e.g. closesocket() instead of just close().

In Linux/Unix a socket descriptor can be treated the same as a file descriptor; not so for SAPI or Winsock.

For Winsock the WSAStartup() and WSACleanup() functions must be called; make it conditional for portable code.

The "socket set" is defined differently for SAPI/Winsock on the one hand and LINUX/UNIX on the other. Always use the FD_* macros for portable code.

Function Reference

The following sections describe the individual functions of the SOCKETS API.

accept

Accepts a connection on a socket.

C syntax

SOCKET accept (SOCKET *so*, struct sockaddr **psAddress,* int *p*i*AddressLen*);

Parameters

so

A descriptor identifying a socket which is listening for connections after a **listen**().

psAddress

An optional pointer to a buffer which receives the socket address of the connecting peer.

piAddrLen

An optional pointer to an integer which contains the length of the address *psAddress*.

Remarks

This function extracts the first connection on the queue of pending connections on listening socket *so*, creates a new socket with the same properties as *so* and returns a handle to the new socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, **accept()** blocks the caller until a connection is

present. If the socket is marked non-blocking and no pending connections are present on the queue, **accept()** returns an error as described below. Socket *so* remains listening.

The argument *psAddress* is a result parameter that is filled in with the socket address of the connecting peer. The p*iAddressLen* is a value-result parameter; it should initially contain the amount of space pointed to by *psAddress*; on return it will contain the actual length (in bytes) of the socket address returned. This call is used with the connection-based SOCK_STREAM socket type. If *psAddress* and/or p*iAddressLen* are equal to NULL, then no information about the remote peer socket address of the accepted socket is returned.

Return Value

If no error occurs, **accept**() returns a value of type SOCKET which is a descriptor for the accepted packet. Otherwise, a value of INVALID_SOCKET is returned, and a specific error code is returned in **errno**.

The integer referred to by *iAddressLen* initially contains the amount of space pointed to by *psAddress*. On return it will contain the actual length in bytes of the socket address returned.

Error Codes

ENETDOWN The network subsystem has failed.

EFAULT The *p*iAddressLen* argument is too small (less than the sizeof a **struct sockaddr**).

EINVAL **listen()** was not invoked prior to **accept()**.

EMFILE The queue is empty upon entry to **accept()** and there are no descriptors available.

ENOBUFS No buffer space is available.

ENOTSOCK The descriptor is not a socket.

EOPNOTSUPP The referenced socket is not a type that supports connection-oriented service.

EWOULDBLOCK The socket is marked as non-blocking and no connections are present to be accepted.

See Also

bind(), connect(), listen(), select(), socket()

bind

Associates a local socket address with a socket.

C syntax

int bind (SOCKET *so,* const struct sockaddr * *psAddress*, int *iAddressLen*);

Parameters

so

A descriptor identifying an unbound socket.

psAddress

The socket address to assign to the socket. The sockaddr structure is defined as follows:

```
struct sockaddr {
        u_short    sa_family;
        char       sa_data[14];
};
```

iAddressLen
The length of the name *psAddress*.

Remarks

This routine is used on an unconnected datagram or stream socket, before subsequent **connect**()s or **listen**()s. When a socket is created with **socket**(), it exists in a name space (address family), but it has no socket address assigned. **bind**() establishes the local association (host address/port number) of the socket by assigning a local address to an unnamed socket.

In the Internet address family, an address consists of several components. For SOCK_DGRAM and SOCK_STREAM, the address consists of three parts: a host address, the protocol number (set implicitly to UDP or TCP, respectively), and a port number which identifies the application. If an application does not care what address is assigned to it, it may specify an Internet address equal to INADDR_ANY, a port equal to 0, or both. If the Internet address is equal to INADDR_ANY, any appropriate network interface will be used; this simplifies application programming in the presence of multi-homed hosts. If the port is specified as 0, SOCKETS will assign a unique port to the application. The application may use **getsockname()** after **bind()** to learn the address that has been assigned to it, but note that **getsockname()** will not necessarily fill in the Internet address until the socket is connected, since several Internet addresses may be valid if the host is multi-homed.

Return Value

If no error occurs, **bind()** returns 0. Otherwise, it returns SOCKET_ERROR, and a specific error code is returned in **errno**.

Error Codes

ENETDOWN SOCKETS has detected that the network subsystem has failed.

EADDRINUSE The specified address is already in use. (See the SO_REUSEADDR socket option under **setsockopt()**.)

EFAULT The *iAddressLen* argument is too small (less than the size of a struct sockaddr).

EAFNOSUPPORT The specified address family is not supported by this protocol.

EINVAL The socket is already bound to an address.

ENOBUFS Not enough buffers available, too many connections.

ENOTSOCK The descriptor is not a socket.

See Also

connect(), listen(), getsockname(), setsockopt(), socket().

closesocket

Closes a socket.

C syntax

int closesocket (SOCKET *so*);

Parameters

so

A descriptor identifying a socket.

Remarks

This function closes a socket. More precisely, it releases the socket descriptor *so*, so that further references to *so* will fail with the error ENOTSOCK. If this is the last reference to the underlying socket, the associated naming information and queued data are discarded.

The semantics of **closesocket**() are affected by the socket options SO_LINGER and SO_DONTLINGER as follows:

Option	Interval	Type of close	Wait for close?
SO_DONTLINGER	Don't care	Graceful	No
SO_LINGER	Zero	Hard	No
SO_LINGER	Non-zero	Graceful	Yes

If SO_LINGER is set (i.e. the *l_onoff* field of the linger structure is non-zero) with a zero timeout interval (*l_linger* is zero), **closesocket()** is not blocked even if queued data has not yet been sent or acknowledged. This is called a "hard" or "abortive" close, because the socket's virtual circuit is reset immediately, and any unsent data is lost. Any **recv()** call on the remote side of the circuit will fail with ECONNRESET.

If SO_LINGER is set with a non-zero timeout interval, the **closesocket()** call blocks until the remaining data has been sent or until the timeout expires. This is called a graceful disconnect. Note that if the socket is set to non-blocking and SO_LINGER is set to a non-zero timeout, the call to **closesocket()** will fail with an error of EWOULDBLOCK.

If SO_DONTLINGER is set on a stream socket (i.e. the *l_onoff* field of the linger structure is zero), the **closesocket()** call will return immediately. However, any data queued for transmission will be sent if possible before the underlying socket is closed. This is also called a graceful disconnect. Note that in this case SOCKETS may not release the socket and other resources for an arbitrary period, which may affect applications which expect to use all available sockets.

Return Value

If no error occurs, **closesocket()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is returned in errno**.**

Error Codes

ENETDOWN SOCKETS has detected that the network subsystem has failed.

ENOTSOCK The descriptor is not a socket.

EWOULDBLOCK The socket is marked as nonblocking and SO_LINGER is set to a nonzero timeout value.

See Also

accept(), socket(), ioctlsocket(), setsockopt.

connect

Establishes a connection to a peer.

C syntax

int connect (SOCKET *so*, const struct sockaddr * *psAddress*,

int *iAddressLen*);

Parameters

so

A descriptor identifying an unconnected socket.

psAddress

The socket address of the peer to which the socket is to be connected.

iAddressLen

The length of *psAddress*.

Remarks

This function is used to create a connection to the specified foreign socket address. The parameter *so* specifies an unconnected datagram or stream socket. If the socket is unbound, unique values are assigned to the local association by the system, and the socket is marked as bound. Note that if the address field of the *psAddress* structure is all zeroes, **connect()** will return the error EADDRNOTAVAIL.

For stream sockets (type SOCK_STREAM), an active connection is initiated to the foreign host using *psAddress* (an address in the name space of the socket). When the socket call completes successfully, the socket is ready to send/receive data.

For a datagram socket (type SOCK_DGRAM), a default destination is set, which will be used on subsequent **send**() and **recv**() calls.

Return Value

If no error occurs, **connect()** returns 0. Otherwise, it returns SOCKET_ERROR, and a specific error code is returned in errno**.**

On a blocking socket, the return value indicates success or failure of the connection attempt.

On a non-blocking socket, if the return value is SOCKET_ERROR and **errno** indicates an error code of EWOULDBLOCK, then your application can either:

1. Use **select()** to determine the completion of the connection request by checking if the socket is writeable, or

2. Use **recv()** until either no error or an error of EWOULDBLOCK is returned.

Error Codes

ENETDOWN SOCKETS has detected that the network subsystem has failed.

EADDRINUSE The specified address is already in use.

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

EAFNOSUPPORT Addresses in the specified family cannot be used with this socket.

ECONNREFUSED The attempt to connect was forcefully rejected.

EDESTADDREQ A destination address is required.

EFAULT The *iAddressLen* argument is incorrect.

EINVAL The socket is not already bound to an address.

EISCONN The socket is already connected.

EMFILE No more file descriptors are available.

ENETUNREACH The network can't be reached from this host at this time.

ENOBUFS No buffer space is available. The socket cannot be connected.

ENOTSOCK The descriptor is not a socket.

ETIMEDOUT Attempt to connect timed out without establishing a connection

EWOULDBLOCK The socket is marked as non-blocking and the connection cannot be completed immediately. It is possible to **select()** the socket while it is connecting by **select()**ing it for writing.

See Also

accept(), bind(), getsockname(), socket() and select().getpeername

Gets the socket address of the peer to which a socket is connected.

C syntax

int getpeername (SOCKET *so*, struct sockaddr * *psAddress*, int * *piAddressLen*);

Parameters

so

A descriptor identifying a connected socket.

psAddress

The structure which is to receive the socket address of the peer.

*pi*AddressLen

A pointer to the size of the *psAddress* structure.

Remarks

getpeername() retrieves the socket address of the peer connected to the socket *so* and stores it in the struct sockaddr identified by *psAddress*. It is used on a connected datagram or stream socket.

On return, the *piAddressLen* argument contains the actual size of the socket address returned in bytes.

Return Value

If no error occurs, **getpeername()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is returned in **errno.**

Error Codes

ENETDOWN SOCKETS has detected that the network subsystem has failed.

EFAULT The **piAddressLen* argument is not large enough.

ENOTCONN The socket is not connected.

ENOTSOCK The descriptor is not a socket.

See Also

bind(), socket(), getsockname().

freeaddrinfo

Function to free memory allocated by the **getaddrinfo()** function.

C syntax

void freeaddrinfo(struct addrinfo **psAI*);

Parameters

psAI

Pointer to a chain of addrinfo structures.

Remarks

The addrinfo structure pointed to by the *psAI* argument is freed, along with any dynamic storage pointed to by the structure. This operation is repeated until a NULL ai_next pointer is encountered.

Return Value

None**.**

See Also

getaddrinfo().

gai_strerror

Gets description of error code returned by **getaddrinfo()**.

C syntax

char *gai_strerror(int iErrcode);

Parameters

iErrcode

Error code returned by **getaddrinfo()**.

Remarks

Use to aid applications in printing error messages based on the EAI_xxx codes returned by **getaddrinfo()**.If the argument is not one of the EAI_xxx values, the function still returns a pointer to a string whose contents indicate an unknown error.

Return Value

Null-terminated string describing the error**.**

See Also

gethostbyname(),

---

getaddrinfo

Gets address information corresponding to an IPv4 and/or IPv6 nodename and servicename.

C syntax

int getaddrinfo(const char *pszN*odename*, const char *p*szServname*,

const struct addrinfo *psH*ints*,struct addrinfo **ppsR*es*);

Parameters

pszNodename

A pointer to a node name or NULL.

pszServname

A pointer to a service name or NULL.

psHints

Optional pointer to structure containing hints.

ppsRes

Pointer to a pointer through which a chain of addrinfo structures will be returned.

Remarks

The *pszNodename* and *pszServname* arguments are pointers to null-terminated strings or NULL. One or both of these two arguments must be a non-NULL pointer. In the normal client scenario, both *pszNodename* and *pszServname* are specified. In the normal server scenario, only *pszServname* is specified. A non-NULL nodename string can be either a node name or a numeric host address string (i.e., a dotted-decimal IPv4 address or an IPv6 hex address). A non-NULL servname string can be either a service name or a decimal port number.

The caller can optionally pass an addrinfo structure, pointed to by *psHints*, to provide hints concerning the type of socket that the caller supports. In this hints structure all members other than ai_flags, ai_family, ai_socktype, and ai_protocol must be zero or a NULL pointer. A value of PF_UNSPEC for ai_family means the caller will accept any protocol family. A value of 0 for ai_socktype means the caller will accept any socket type. A value of 0 for ai_protocol means the caller will accept any protocol. For example, if the caller handles only TCP and not UDP, then the ai_socktype member of the hints structure should be set to SOCK_STREAM when getaddrinfo() is called. If the caller handles only IPv4 and not IPv6, then the ai_family member of the hints structure should be set to PF_INET when getaddrinfo() is called. If the third argument to getaddrinfo() is a NULL pointer, this is the same as if the caller had filled in an addrinfo structure initialized to zero with ai_family set to PF_UNSPEC.

The addrinfo structure is defined as:

struct addrinfo {

```
int                ai_flags;       /* AI_PASSIVE, AI_CANONNAME,
                                      AI_NUMERICHOST */
int                ai_family;      /* PF_xxx */
int                ai_socktype;    /* SOCK_xxx */
int                ai_protocol;    /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
size_t             ai_addrlen;     /* length of ai_addr */
char               *ai_canonname;  /* canonical name for nodename */
struct sockaddr    *ai_addr;       /* binary address */
struct addrinfo    *ai_next;       /* next structure in linked list */
};
```

Return Value

The return value from the function is 0 upon success or a nonzero error code.

Upon successful return a pointer to a linked list of one or more addrinfo structures is returned through the *ppsRes* argument. The caller can process each addrinfo structure in this list by following the ai_next pointer, until a NULL pointer is encountered. In each returned addrinfo structure the three members ai_family, ai_socktype, and ai_protocol are the corresponding arguments for a call to the **socket()** function. In each addrinfo structure the ai_addr member points to a filled-in socket address structure whose length is specified by the ai_addrlen member.

If the AI_PASSIVE bit is set in the ai_flags member of the hints structure, then the caller plans to use the returned socket address structure in a call to **bind()**. In this case, if the nodename argument is a NULL pointer, then the IP address portion of the socket address structure will be set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address. If the AI_PASSIVE bit is not set in the ai_flags member of the hints structure, then the returned socket address structure will be ready for a call to **connect()** (for a connection-oriented protocol) or either **connect()** or **sendto()** (for a connectionless protocol). In this case, if the nodename argument is a NULL pointer, then the IP address portion of the socket address structure will be set to the loopback address.

If the AI_CANONNAME bit is set in the ai_flags member of the hints structure, then upon successful return the ai_canonname member of the first addrinfo structure in the linked list will point to a null-terminated string containing the canonical name of the specified nodename.

If the AI_NUMERICHOST bit is set in the ai_flags member of the hints structure, then a non-NULL nodename string must be a numeric host address string. Otherwise an error of EAI_NONAME is returned. This flag prevents any type of name resolution service (e.g., the DNS) from being called.

All of the information returned by **getaddrinfo()** is dynamically allocated: the addrinfo structures, and the socket address structures and canonical node name strings pointed to by the addrinfo structures. To return this information to the system the function **freeaddrinfo()** is called:

Error Codes

The following names are the nonzero error codes from **getaddrinfo**():

EAI_ADDRFAMILY address family for nodename not supported

EAI_AGAIN temporary failure in name resolution

EAI_BADFLAGS	invalid value for ai_flags
EAI_FAIL	non-recoverable failure in name resolution
EAI_FAMILY	ai_family not supported
EAI_MEMORY	memory allocation failure
EAI_NODATA	no address associated with nodename
EAI_NONAME	nodename nor servname provided, or not known
EAI_SERVICE	servname not supported for ai_socktype
EAI_SOCKTYPE	ai_socktype not supported
EAI_SYSTEM	system error returned in errno

To aid applications in printing error messages based on the EAI_xxx codes. use the **gai_strerror()** function.

See Also

freeaddrinfo(), gai_strerror(), gethostbyname(), socket(), connect(), bind(), sendto().

gethostbyaddr

Gets host information corresponding to an IPv4 address.

C syntax

struct hostent * gethostbyaddr (const char * *pcAddr*, int *len*, int *type*);

Parameters

pcAddr

A pointer to an address in network byte order.

len

The length of the address, which must be 4 for PF_INET addresses.

type

The type of the address, which must be PF_INET.

Remarks

gethostbyaddr() always returns a zero with an error code of WSANO_DATA

Return Value

Returns a NULL pointer and WSANO_DATA in errno**.**

Error Codes

WSANO_DATA	Valid name, no data record of requested type.

See Also

gethostbyname(),

gethostbyname

Gets host information corresponding to a hostname.

C syntax

struct hostent * gethostbyname (const char * *pszName*);

Parameters

pszName

A pointer to the name of the host.

Remarks

gethostbyname () returns a pointer to the following structure which contains the name(s) and address which correspond to the hostname *pszName*.

.

```
struct hostent {
    char *  h_name;
    char ** h_aliases;
    short   h_addrtype;
    short   h_length;
    char ** h_addr_list;
};
```

The members of this structure are:

Element	Usage
h_name	Official name of the host (PC).
h_aliases	A NULL-terminated array of alternate names.
h_addrtype	The type of address being returned; for SOCKETS this is always PF_INET.
h_length	The length, in bytes, of each address; for PF_INET, this is always 4.
h_addr_list	A NULL-terminated list of addresses for the host. Addresses are returned in network byte order.

The macro h_addr is defined to be h_addr_list[0] for compatibility with older software.

The pointer which is returned points to a structure which is allocated by SOCKETS. The application must never attempt to modify this structure or to free any of its components. The application should copy any information which it needs before issuing any other SOCKETS API calls.

A **gethostbyname()** implementation must not resolve IP address strings passed to it. Such a request should be treated exactly as if an unknown host name were passed. An application with an IP address string to resolve should use **inet_addr()** to convert the string to an IP address.

Return Value

If no error occurs, **gethostbyname()** returns a pointer to the hostent structure described above. Otherwise it returns a NULL pointer and a specific error number is returned in errno.

Error Codes

ENETDOWN SOCKETS has detected that the network subsystem has failed.

WSAHOST_NOT_FOUND Authoritative Answer Host not found.

WSATRY_AGAIN Non-Authoritative Host not found, or SERVERFAIL.

WSANO_RECOVERY Non recoverable errors, FORMERR, REFUSED, NOTIMP.

WSANO_DATA Valid name, no data record of requested type.

See Also

getaddrinfo().

---

gethostname

Return the standard host name for the local machine.

C syntax

int gethostname (char * *pszName*, int *iAddressLen*);

Parameters

pszName

A pointer to a buffer that will receive the host name.

iAddressLen

The length of the buffer.

Remarks

This routine returns the name of the local host into the buffer specified by the *pszName* parameter. The host name is returned as a null-terminated string. The form of the host name is dependent on the SOCKETS configuration file. However, it is guaranteed that the name returned will be successfully parsed by **gethostbyname().**

Return Value

If no error occurs, **gethostname()** returns 0, otherwise it returns SOCKET_ERROR and a specific error code is returned in errno.

Error Codes

EFAULT The *iAddressLen* parameter is too small

ENETDOWN SOCKETS has detected that the network subsystem has failed.

See Also

gethostbyname().

---

getprotobyname

Gets protocol information corresponding to a protocol name.

C syntax

struct protoent * getprotobyname (const char * *pszName*);

Parameters

pszName

A pointer to a protocol name.

Remarks

getprotobyname() returns a pointer to the following structure which contains the name(s) and protocol number which correspond to the given protocol *pszName*.

```
struct protoent {
    char *  p_name;
    char ** p_aliases;
    short   p_proto;
};
```

The members of this structure are:

Element	Usage
p_name	Official name of the protocol.
p_aliases	A NULL-terminated array of alternate names.
p_proto	The protocol number, in host byte order.

The pointer which is returned points to a structure which is allocated by the SOCKETS library. The application must never attempt to modify this structure or to free any of its components. The application should copy any information which it needs before issuing any other SOCKETS API calls.

Return Value

If no error occurs, **getprotobyname()** returns a pointer to the protoent structure described above. Otherwise it returns a NULL pointer and a specific error number is returned in errno.

Error Codes

ENETDOWN	SOCKETS has detected that the network subsystem has failed.
WSANO_RECOVERY	Non recoverable errors, FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.

See Also

getaddrinfo(), getprotobynumber()

getprotobynumber

Gets protocol information corresponding to a protocol number.

C syntax

struct protoent * getprotobynumber (int *number*);

Parameters

number

A protocol number, in host byte order.

Remarks

This function returns a pointer to a protoent structure as described above in **getprotobyname()**. The contents of the structure correspond to the given protocol number.

The pointer which is returned points to a structure which is allocated by SOCKETS. The application must never attempt to modify this structure or to free any of its components. The application should copy any information which it needs before issuing any other SOCKETS API calls.

Return Value

If no error occurs, **getprotobynumber()** returns a pointer to the protoent structure described above. Otherwise it returns a NULL pointer and a specific error number is returned in errno.

Error Codes

ENETDOWN	SOCKETS has detected that the network subsystem has failed.
WSANO_RECOVERY	Non recoverable errors, FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.

See Also

getaddrinfo(), getprotobyname()

getservbyname

Gets service information corresponding to a service name and protocol.

struct servent * getservbyname (const char * *pszName*,

const char * *proto*);

Parameters

pszName

A pointer to a service name.

proto

An optional pointer to a protocol name. If this is NULL, **getservbyname()** returns the first service entry for which the *pszName* matches the s_name or one of the s_aliases. Otherwise **getservbyname()** matches both the *pszName* and the *proto*.

Remarks

getservbyname() returns a pointer to the following structure which contains the name(s) and service number which correspond to the given service *pszName*.

```
struct servent {
    char *  s_name;
    char ** s_aliases;
    short   s_port;
    char *  s_proto;
```

```
};
```

The members of this structure are:

Element	Usage
s_name	Official name of the service.
s_aliases	A NULL-terminated array of alternate names.
s_port	The port number at which the service may be contacted. Port numbers are returned in network byte order.
s_proto	The name of the protocol to use when contacting the service.

The pointer which is returned points to a structure which is allocated by the SOCKETS library. The application must never attempt to modify this structure or to free any of its components. The application should copy any information which it needs before issuing any other SOCKETS API calls.

Return Value

If no error occurs, **getservbyname()** returns a pointer to the servent structure described above. Otherwise it returns a NULL pointer and a specific error number is returned in errno**.**

Error Codes

ENETDOWN SOCKETS has detected that the network subsystem has failed.

WSANO_RECOVERY Non recoverable errors, FORMERR, REFUSED, NOTIMP.

WSANO_DATA Valid name, no data record of requested type.

See Also

getaddrinfo(), getservbyport()

getservbyport

Gets service information corresponding to a port and protocol.

C syntax

struct servent * getservbyport (int *port*, const char * *proto*);

Parameters

port

The port for a service, in network byte order.

proto

An optional pointer to a protocol name. If this is NULL, **getservbyport()** returns the first service entry for which the *port* matches the s_port. Otherwise **getservbyport()** matches both the *port* and the *proto*.

Remarks

getservbyport() returns a pointer a servent structure as described above for **getservbyname()**.

The pointer which is returned points to a structure which is allocated by SOCKETS. The application must never attempt to modify this structure or to free any of its components. The application should copy any information which it needs before issuing any other SOCKETS API calls.

Return Value

If no error occurs, **getservbyport()** returns a pointer to the servent structure described above. Otherwise it returns a NULL pointer and a specific error number is returned in errno**.**

Error Codes

ENETDOWN SOCKETS has detected that the network subsystem has failed.

WSANO_RECOVERY Non recoverable errors, FORMERR, REFUSED, NOTIMP.

WSANO_DATA Valid name, no data record of requested type.

See Also

getaddrinfo(), getservbyname()

---

getsockname

Gets the local socket address for a socket.

C syntax

int getsockname (SOCKET *so*, struct sockaddr * *psAddress*,

int *** p*iAddressLen* **);

Parameters

so

A descriptor identifying a bound socket.

psAddress

Receives the socket address (name) of the socket.

*p*iAddressLen

A pointer to the size of the *psAddress* buffer.

Remarks

getsockname() retrieves the current socket address for the specified socket descriptor in *psAddress*. It is used on a bound and/or connected socket specified by the *so* parameter. The local association is returned. This call is especially useful when a **connect()** call has been made without doing a **bind()** first; this call provides the only means by which you can determine the local association which has been set by the system.

On return, the *piAddressLen* argument contains the actual size of the socket address returned in bytes.

If a socket was bound to INADDR_ANY, indicating that any of the host's IP addresses should be used for the socket, **getsockname()** will not necessarily return information about the host IP address, unless the socket has been connected with **connect()** or **accept().** A SOCKETS application must not assume that the IP address will be changed from INADDR_ANY unless the socket is connected. This is because for a multi-homed

host the IP address that will be used for the socket is unknown unless the socket is connected.

Return Value

If no error occurs, **getsockname()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is returned in **errno.**

Error Codes

ENETDOWN SOCKETS has detected that the network subsystem has failed.

EFAULT The **piAddressLen* argument is not large enough.

ENOTSOCK The descriptor is not a socket.

EINVAL The socket has not been bound to an address with **bind().**

See Also

bind(), socket(), getpeername().

getsockopt

Retrieves a socket option.

C syntax

int getsockopt (SOCKET *so*, int *iLevel*, int *iOptname*,

char * pcO*ptval*, int * *piOptlen*);

Parameters

so

A descriptor identifying a socket.

iLevel

The level at which the option is defined; the only supported levels are SOL_SOCKET and IPPROTO_TCP.

iOptname

The socket option for which the value is to be retrieved.

pcOptval

A pointer to the buffer in which the value for the requested option is to be returned.

piOptlen

A pointer to the size of the *pcOptval* buffer.

Remarks

getsockopt() retrieves the current value for a socket option associated with a socket of any type, in any state, and stores the result in pcO*ptval*. Options may exist at multiple protocol levels, but they are always present at the uppermost "socket" level. Options affect socket operations, such as whether an operation blocks or not, the routing of packets, out-of-band data transfer, etc.

The value associated with the selected option is returned in the buffer pcO*ptval*. The integer pointed to by piO*ptlen* should originally contain the size of this buffer; on return,

it will be set to the size of the value returned. For SO_LINGER, this will be the size of a struct linger; for all other options it will be the size of an integer.

If the option was never set with **setsockopt(),** then **getsockopt()** returns the default value for the option.

The following options are supported for **getsockopt()**. The Type identifies the type of data addressed by *optval*. The TCP_NODELAY option uses *level* IPPROTO_TCP; all other options use *level* SOL_SOCKET.

Value	Type	Meaning	Default
SO_ACCEPTCONN	BOOL	Socket is listen()ing.	FALSE
SO_BROADCAST	BOOL	Socket is configured for the transmission of broadcast messages.	FALSE
SO_DEBUG	BOOL	Debugging is enabled.	FALSE
SO_DONTLINGER	BOOL	If true, the SO_LINGER option is disabled.	TRUE
SO_DONTROUTE	BOOL	Routing is disabled.	FALSE
SO_ERROR	int	Retrieve error status and clear.	0
SO_KEEPALIVE	BOOL	Keepalives are being sent.	FALSE
SO_LINGER	struct linger *	Returns the current linger options.	l_onoff is 0
SO_OOBINLINE	BOOL	Out-of-band data is being received in the normal data stream.	FALSE
SO_RCVBUF	int	Buffer size for receives	1460
SO_REUSEADDR	BOOL	The socket may be bound to an address which is already in use.	FALSE
SO_SNDBUF	int	Buffer size for sends	1460
SO_TYPE	int	The type of the socket (e.g. SOCK_STREAM).	As created
TCP_NODELAY	BOOL	Disables the Nagle algorithm for send coalescing.	FALSE

Calling **getsockopt()** with an unsupported option will result in an error code of ENOPROTOOPT being returned from **WSAGetLastError()**.

Return Value

If no error occurs, **getsockopt()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is returned in errno.

Error Codes

ENETDOWN SOCKETS has detected that the network subsystem has failed.

EFAULT The *piOptlen* argument was invalid.

ENOPROTOOPT The option is unknown or unsupported. In particular, SO_BROADCAST is not supported on sockets of type SOCK_STREAM, while SO_ACCEPTCONN, SO_DONTLINGER, SO_KEEPALIVE, SO_LINGER and SO_OOBINLINE are not supported on sockets of type SOCK_DGRAM.

ENOTSOCK The descriptor is not a socket.

See Also

setsockopt(), socket().

htonl

Converts a **u_long** from host to network byte order.

C syntax

u_long htonl (u_long *ulHostlong*);

Parameters

ulHostlong

A 32-bit number in host byte order.

Remarks

This routine takes a 32-bit number in host byte order and returns a 32-bit number in network byte order.

Return Value

htonl() returns the value in network byte order.

See Also

htons(), ntohl(), ntohs().

htons

Converts a **u_short** from host to network byte order.

C syntax

u_short htons (u_short usH*ostshort*);

Parameters

usHostshort

A 16-bit number in host byte order.

Remarks

This routine takes a 16-bit number in host byte order and returns a 16-bit number in network byte order.

Return Value

htons() returns the value in network byte order.

See Also

htonl(), ntohl(), ntohs().

inet_addr

Converts a string containing a dotted decimal IPv4 address into an **in_addr**.

C syntax

unsigned long inet_addr (const char * *pc*);

Parameters

pc

A character string representing a number expressed in the Internet standard "." notation.

Remarks

This function interprets the character string specified by the *pc* parameter. This string represents a numeric Internet address expressed in the Internet standard "." notation. The value returned is a number suitable for use as an Internet address. All Internet addresses are returned in network order (bytes ordered from left to right).

Internet Addresses

Values specified using the "." notation take one of the following forms:

```
a.b.c.d     a.b.c   a.b     a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. Note that when an Internet address is viewed as a 32-bit integer quantity on the Intel architecture, the bytes referred to above appear as "d.c.b.a". That is, the bytes on an Intel processor are ordered from right to left.

Note: The following notations are only used by Berkeley, and nowhere else on the Internet. In the interests of compatibility with their software, they are supported as specified.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as "128.net.host".

When a two part address is specified, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as "net.host".

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

Return Value

If no error occurs, **inet_addr()** returns an unsigned long containing a suitable binary representation of the Internet address given. If the passed-in string does not contain a legitimate Internet address, for example if a portion of an "a.b.c.d" address exceeds 255, **inet_addr()** returns the value INADDR_NONE.

See Also

inet_ntoa(), getaddrinfo()

inet_ntoa

Converts a network IPv4 address into a string in dotted format.

C syntax

char * inet_ntoa (struct in_addr *sIn*);

Parameters

sIn

A structure which represents an Internet host address.

Remarks

This function takes an Internet address structure specified by the *sIn* parameter. It returns an ASCII string representing the address in "." notation as "a.b.c.d". Note that the string returned by **inet_ntoa()** resides in memory which is allocated by SOCKETS. The application should not make any assumptions about the way in which the memory is allocated. The data is guaranteed to be valid until the next SOCKETS API call, but no longer.

Return Value

If no error occurs, **inet_ntoa()** returns a char pointer to a static buffer containing the text address in standard "." notation. Otherwise, it returns NULL. The data should be copied before another SOCKETS call is made.

See Also

inet_addr(), inet_ntop().

ioctlsocket

Controls the mode of a socket.

C syntax

int ioctlsocket (SOCKET *so*, long *lCmd*, u_long * *pulArgp*);

Parameters

so

A descriptor identifying a socket.

lCmd

The command to perform on the socket *so*.

pulArgp

A pointer to a parameter for *lCmd*.

Remarks

This routine may be used on any socket in any state. It is used to get or retrieve operating parameters associated with the socket, independent of the protocol and communications subsystem. The following commands are supported:

Command Semantics

FIONBIO Enable or disable non-blocking mode on the socket *so*. *pulArgp* points at an **unsigned long**, which is non-zero if non-blocking mode is to be enabled and zero if it is to be disabled. When a socket is created, it operates in blocking mode (i.e. non-blocking mode is disabled). This is consistent with BSD sockets.

FIONREAD Determine the amount of data which can be read atomically from socket *so*. *pulArgp* points at an **unsigned long** in which **ioctlsocket()** stores the result. If *so* is of type SOCK_STREAM, FIONREAD returns the total amount of data which may be read in a single **recv()**; this is normally the same as the total amount of data queued on the socket. If *so* is of type SOCK_DGRAM, FIONREAD returns the size of the first datagram queued on the socket.

SIOCATMARKDetermine whether or not all out-of-band data has been read. This applies only to a socket of type SOCK_STREAM which has been configured for in-line reception of any out-of-band data (SO_OOBINLINE). If no out-of-band data is waiting to be read, the operation returns TRUE. Otherwise it returns FALSE, and the next **recv()** or **recvfrom()** performed on the socket will retrieve some or all of the data preceding the "mark"; the application should use the SIOCATMARK operation to determine whether any remains. If there is any normal data preceding the "urgent" (out of band) data, it will be received in order. (Note that a **recv()** or **recvfrom()** will never mix out-of-band and normal data in the same call.) *argp* points at a **BOOL** in which **ioctlsocket()** stores the result.

Compatibility

This function is a subset of **ioctl()** as used in Berkeley sockets. In particular, there is no command which is equivalent to FIOASYNC, while SIOCATMARK is the only socket-level command which is supported.

Return Value

Upon successful completion, the **ioctlsocket()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is returned in errno**.**

Error Codes

ENETDOWN SOCKETS has detected that the network subsystem has failed.

EINVAL *lCmd* is not a valid command, or pulA*rgp* is not an acceptable parameter for *lCmd*, or the command is not applicable to the type of socket supplied

ENOTSOCK The descriptor *so* is not a socket.

See Also

socket(), setsockopt(), getsockopt().

listen

Establishes a socket to listen for incoming connection.

C syntax

int listen (SOCKET *so*, int *iBacklog*);

Parameters

so

A descriptor identifying a bound, unconnected socket.

iBacklog
The maximum length to which the queue of pending connections may grow.

Remarks

To accept connections, a socket is first created with **socket**(), a backlog for incoming connections is specified with **listen**(), and then the connections are accepted with **accept**(). **listen**() applies only to sockets that support connections, i.e. those of type SOCK_STREAM. The socket *so* is put into "passive" mode where incoming connections are acknowledged and queued pending acceptance by the process.

This function is typically used by servers that could have more than one connection request at a time: if a connection request arrives with the queue full, the client will receive an error with an indication of ECONNREFUSED.

Compatibility

iBacklog is limited (silently) to 5. As in 4.3BSD, illegal values (less than 1 or greater than 5) are replaced by the nearest legal value.

Return Value

If no error occurs, **listen()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is returned in errno.

Error Codes

ENETDOWN SOCKETS has detected that the network subsystem has failed.

EADDRINUSE An attempt has been made to **listen()** on an address in use.

EINVAL The socket has not been bound with **bind()** or is already connected.

EISCONN The socket is already connected.

EMFILE No more file descriptors are available.

ENOBUFS No buffer space is available.

ENOTSOCK The descriptor is not a socket.

EOPNOTSUPP The referenced socket is not of a type that supports the **listen()** operation.

See Also

accept(), connect(), socket().

ntohl

Converts a **u_long** from network to host byte order.

C syntax

u_long ntohl (u_long *ulNetlong*);

Parameters

ulNetlong
A 32-bit number in network byte order.

Remarks

This routine takes a 32-bit number in network byte order and returns a 32-bit number in host byte order.

Return Value

ntohl() returns the value in host byte order.

See Also

htonl(), htons(), ntohs().

---

ntohs

Converts a **u_short** from network to host byte order.

C syntax

u_short ntohs (u_short *usNetshort*);

Parameters

usNetshort

A 16-bit number in network byte order.

Remarks

This routine takes a 16-bit number in network byte order and returns a 16-bit number in host byte order.

Return Value

ntohs() returns the value in host byte order.

See Also

htonl(), htons(), ntohl().

---

recv

Receives data from a socket.

C syntax

int recv (SOCKET *so*, char * pc*buf*, int *iLen*, int *iFlags*);

Parameters

so

A descriptor identifying a connected socket.

pcBuf

A buffer for the incoming data.

iLen

The length of pcB*uf*.

iFlags

Specifies the way in which the call is made.

Remarks

This function is used on connected datagram or stream sockets specified by the *so* parameter and is used to read incoming data.

For sockets of type SOCK_STREAM, as much information as is currently available up to the size of the buffer supplied is returned. If the socket has been configured for in-line reception of out-of-band data (socket option SO_OOBINLINE) and out-of-band data is unread, only out-of-band data will be returned. The application may use the **ioctlsocket()** SIOCATMARK to determine whether any more out-of-band data remains to be read.

For datagram sockets, data is extracted from the first enqueued datagram, up to the size of the buffer supplied. If the datagram is larger than the buffer supplied, the buffer is filled with the first part of the datagram, the excess data is lost, and **recv()** returns the error EMSGSIZE.

If no incoming data is available at the socket, the **recv()** call waits for data to arrive unless the socket is non-blocking. In this case a value of SOCKET_ERROR is returned with the error code set to EWOULDBLOCK. The **select**() call may be used to determine when more data arrives.

If the socket is of type SOCK_STREAM and the remote side has shut down the connection gracefully, a **recv()** will complete immediately with 0 bytes received. If the connection has been reset, a **recv()** will fail with the error ECONNRESET.

iFlags may be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the i*Flags* parameter. The latter is constructed by or-ing any of the following values:

Value	Meaning
MSG_PEEK	Peek at the incoming data. The data is copied into the buffer but is not removed from the input queue.
MSG_OOB	Process out-of-band data.

Return Value

If no error occurs, **recv()** returns the number of bytes received. If the connection has been closed, it returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is returned in errno.

Error Codes

ENETDOWN SOCKETS has detected that the network subsystem has failed.

ENOTCONN The socket is not connected.

ENOTSOCK The descriptor is not a socket.

EOPNOTSUPP MSG_OOB was specified, but the socket is not of type SOCK_STREAM.

ESHUTDOWN The socket has been shutdown; it is not possible to **recv()** on a socket after **shutdown()** has been invoked with *how* set to 0 or 2.

EWOULDBLOCK The socket is marked as non-blocking and the receive operation would block.

EMSGSIZE The datagram was too large to fit into the specified buffer and was truncated.

EINVAL The socket has not been bound with **bind().**

ECONNABORTED The virtual circuit was aborted due to timeout or other failure.

ECONNRESET The virtual circuit was reset by the remote side.

See Also

recvfrom(), ,recv(), send(), select(), socket()

recvfrom

Receives a datagram and store the source address.

C syntax

int recvfrom (SOCKET *so*, char * pcB*uf*, int iL*en*, int iF*lags*,

struct sockaddr * psF*rom*, int * piF*romlen*);

Parameters

so

A descriptor identifying a bound socket.

pcBuf

A buffer for the incoming data.

iLen

The length of pcB*uf*.

iFlags

Specifies the way in which the call is made.

psFrom

An optional pointer to a buffer which will hold the source address upon return.

piFromlen

An optional pointer to the size of the psF*rom* buffer.

Remarks

This function is used to read incoming data on a (possibly connected) socket and capture the address from which the data was sent.

For sockets of type SOCK_STREAM, as much information as is currently available up to the size of the buffer supplied is returned. If the socket has been configured for in-line reception of out-of-band data (socket option SO_OOBINLINE) and out-of-band data is unread, only out-of-band data will be returned. The application may use the **ioctlsocket()** SIOCATMARK to determine whether any more out-of-band data remains to be read. The psF*rom* and piF*romlen* parameters are ignored for SOCK_STREAM sockets.

For datagram sockets, data is extracted from the first enqueued datagram, up to the size of the buffer supplied. If the datagram is larger than the buffer supplied, the buffer is filled with the first part of the message, the excess data is lost, and **recvfrom()** returns the error code EMSGSIZE.

If psF*rom* is non-zero, and the socket is of type SOCK_DGRAM, the network address of the peer which sent the data is copied to the corresponding struct sockaddr. The value pointed to by piF*romlen* is initialized to the size of this structure, and is modified on return to indicate the actual size of the address stored there.

If no incoming data is available at the socket, the **recvfrom()** call waits for data to arrive unless the socket is non-blocking. In this case a value of SOCKET_ERROR is returned with the error code set to EWOULDBLOCK. The **select**() call may be used to determine when more data arrives.

If the socket is of type SOCK_STREAM and the remote side has shut down the connection gracefully, a **recvfrom()** will complete immediately with 0 bytes received. If the connection has been reset **recv()** will fail with the error ECONNRESET.

iFlags may be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the i*Flags* parameter. The latter is constructed by or-ing any of the following values:

Value	Meaning
MSG_PEEK	Peek at the incoming data. The data is copied into the buffer but is not removed from the input queue.
MSG_OOB	Process out-of-band data.

Return Value

If no error occurs, **recvfrom()** returns the number of bytes received. If the connection has been closed, it returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is returned in errno.

Error Codes

ENETDOWN SOCKETS has detected that the network subsystem has failed.

EFAULT The pi*Fromlen* argument was invalid: the ps*From* buffer was too small to accommodate the peer address.

EINVAL The socket has not been bound with **bind()**.

ENOTCONN The socket is not connected (SOCK_STREAM only).

ENOTSOCK The descriptor is not a socket.

EOPNOTSUPP MSG_OOB was specified, but the socket is not of type SOCK_STREAM.

ESHUTDOWN The socket has been shutdown; it is not possible to **recvfrom()** on a socket after **shutdown()** has been invoked with *how* set to 0 or 2.

EWOULDBLOCK The socket is marked as non-blocking and the **recvfrom()** operation would block.

EMSGSIZE The datagram was too large to fit into the specified buffer and was truncated.

ECONNABORTED The virtual circuit was aborted due to timeout or other failure.

ECONNRESET The virtual circuit was reset by the remote side.

See Also

recv(), send(), socket().

select

Determines the status of one or more sockets, waiting if necessary.

C syntax

int select (int iN*fds*, fd_set * psR*eadfds*, fd_set * psW*ritefds*,
fd_set * psE*xceptfds*, const struct timeval * psT*imeout*);

Parameters

iNfds

This argument is ignored and included only for the sake of compatibility.

psRadfds

An optional pointer to a set of sockets to be checked for readability.

psWritefds

An optional pointer to a set of sockets to be checked for writability

psExceptfds

An optional pointer to a set of sockets to be checked for errors.

psTimeout

The maximum time for **select()** to wait, or NULL for blocking operation.

Remarks

This function is used to determine the status of one or more sockets. For each socket, the caller may request information on read, write or error status. The set of sockets for which a given status is requested is indicated by an fd_set structure. Upon return, the structure is updated to reflect the subset of these sockets which meet the specified condition, and **select()** returns the number of sockets meeting the conditions. A set of macros is provided for manipulating an fd_set. These macros are compatible with those used in the Berkeley software, but the underlying representation is completely different and the same as that used in Winsock.

The parameter psR*eadfds* identifies those sockets which are to be checked for readability. If the socket is currently **listen()**ing, it will be marked as readable if an incoming connection request has been received, so that an **accept()** is guaranteed to complete without blocking. For other sockets, readability means that queued data is available for reading or, for sockets of type SOCK_STREAM, that the virtual socket corresponding to the socket has been closed, so that a **recv()** or **recvfrom()** is guaranteed to complete without blocking. If the virtual circuit was closed gracefully, then a **recv()** will return immediately with 0 bytes read; if the virtual circuit was reset, then a **recv()** will complete immediately with the error code ECONNRESET. The presence of out-of-band data will be checked if the socket option SO_OOBINLINE has been enabled (see **setsockopt()**).

The parameter psW*ritefds* identifies those sockets which are to be checked for writability. If a socket is **connect()**ing (non-blocking), writability means that the connection establishment successfully completed. If the socket is not in the process of **connect()**ing, writability means that a **send()** or **sendto()** will complete without blocking.

The parameter psE*xceptfds* identifies those sockets which are to be checked for the presence of out-of-band data or any exceptional error conditions. Note that out-of-band data will only be reported in this way if the option SO_OOBINLINE is FALSE. For a SOCK_STREAM, the breaking of the connection by the peer or due to KEEPALIVE failure will be indicated as an exception. If a socket is **connect()**ing (non-blocking), failure of the connect attempt is indicated in *psExceptfds.*

Any of psR*eadfds*, psW*ritefds*, or psE*xceptfds* may be given as NULL if no descriptors are of interest.

Four macros are defined in the header file **socket.h** for manipulating the descriptor sets. The variable FD_SETSIZE determines the maximum number of descriptors in a set. (The default value of FD_SETSIZE is 16, which may be modified by #defining FD_SETSIZE to another value before #including **socket.h**.) Internally, an fd_set is represented as an array of SOCKETs. The macros are:

FD_CLR(so, *psSet)	Removes the descriptor so from set.
FD_ISSET(so, *pSset)	Nonzero if so is a member of the set, zero otherwise.
FD_SET(so, *psSet)	Adds descriptor so to set.
FD_ZERO(*psSet)	Initializes the set to the NULL set.

The parameter psT*imeout* controls how long the **select()** may take to complete. If psT*imeout* is a null pointer, **select()** will block indefinitely until at least one descriptor meets the specified criteria. Otherwise, psT*imeout* points to a struct timeval which specifies the maximum time that **select()** should wait before returning. If the timeval is initialized to {0, 0}, **select()** will return immediately; this is used to "poll" the state of the selected sockets.

Return Value

select() returns the total number of descriptors which are ready and contained in the fd_set structures, 0 if the time limit expired, or SOCKET_ERROR if an error occurred. If the return value is SOCKET_ERROR, **errno** contains the specific error code.

Error Codes

ENETDOWN SOCKETS has detected that the network subsystem has failed.

EINVAL The psT*imeout* value is not valid.

ENOTSOCK One of the descriptor sets contains an entry which is not a socket.

See Also

accept(), connect(), recv(), recvfrom(), send().

send

Sends data on a connected socket.

C syntax

int send (SOCKET *so*, const char * *pcBuf*, int *iLen*, int *iFlags*);

Parameters

so

A descriptor identifying a connected socket.

pcBuf

A buffer containing the data to be transmitted.

iLen

The length of the data in *pcBuf*.

iFlags
Specifies the way in which the call is made.

Remarks

send() is used on connected datagram or stream sockets and is used to write outgoing data on a socket. For datagram sockets, care must be taken not to exceed the maximum IP packet size of the underlying subnets. If the data is too long to pass atomically through the underlying protocol the error EMSGSIZE is returned, and no data is transmitted.

Note that the successful completion of a **send()** does not indicate that the data was successfully delivered.

If no buffer space is available within the transport system to hold the data to be transmitted, **send**() will block unless the socket has been placed in a non-blocking I/O mode. On non-blocking SOCK_STREAM sockets, the number of bytes written may be between 1 and the requested length, depending on buffer availability on both the local and foreign hosts. The **select**() call may be used to determine when it is possible to send more data.

iFlags may be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by or-ing any of the following values:

Value	Meaning
MSG_DONTROUTE	Specifies that the data should not be subject to routing
MSG_OOB	Send out-of-band data (SOCK_STREAM only)

Return Value

If no error occurs, **send()** returns the total number of characters sent. (Note that this may be less than the number indicated by *len*.) Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is returned in errno.

Error Codes

ENETDOWN	SOCKETS has detected that the network subsystem has failed.
EACCES	The requested address is a broadcast address, but the appropriate flag was not set.
EFAULT	The *pcBuf* argument is not in a valid part of the user address space.
ENETRESET	The connection must be reset because SOCKETS dropped it.
ENOBUFS	SOCKETS reports a buffer deadlock.
ENOTCONN	The socket is not connected.
ENOTSOCK	The descriptor is not a socket.
EOPNOTSUPP	MSG_OOB was specified, but the socket is not of type SOCK_STREAM.
ESHUTDOWN	The socket has been shutdown; it is not possible to **send()** on a socket after **shutdown()** has been invoked with how set to 1 or 2.
EWOULDBLOCK	The socket is marked as non-blocking and the requested operation would block.
EMSGSIZE	The socket is of type SOCK_DGRAM, and the datagram is larger than the maximum supported by SOCKETS.
EINVAL	The socket has not been bound with **bind().**

ECONNABORTED	The virtual circuit was aborted due to timeout or other failure.
ECONNRESET	The virtual circuit was reset by the remote side.

See Also

recv(), recvfrom(), socket(), sendto().

sendto

Sends data to a specific destination.

C syntax

int sendto (SOCKET *so*, const char * *pcBuf*, int *iLen*, int *iFlags*,

const struct sockaddr * *psTo*, int *iTolen*);

Parameters

so
A descriptor identifying a socket.

pcBuf
A buffer containing the data to be transmitted.

iLen
The length of the data in *pcBuf*.

iFlags
Specifies the way in which the call is made.

psTo
An optional pointer to the address of the target socket.

iTolen
The size of the address in *to*.

Remarks

sendto() is used on datagram or stream sockets and is used to write outgoing data on a socket. For datagram sockets, care must be taken not to exceed the maximum IP packet size of the underlying subnets. If the data is too long to pass atomically through the underlying protocol the error EMSGSIZE is returned, and no data is transmitted.

Note that the successful completion of a **sendto()** does not indicate that the data was successfully delivered.

sendto() is normally used on a SOCK_DGRAM socket to send a datagram to a specific peer socket identified by the psT*o* parameter. On a SOCK_STREAM socket, the psT*o* and iT*olen* parameters are ignored; in this case the **sendto()** is equivalent to **send**().

To send a broadcast (on a SOCK_DGRAM only), the address in the *to* parameter should be constructed using the special IP address INADDR_BROADCAST (defined in **socket.h**) together with the intended port number. It is generally inadvisable for a broadcast datagram to exceed the size at which fragmentation may occur, which implies that the data portion of the datagram (excluding headers) should not exceed 512 bytes.

If no buffer space is available within the transport system to hold the data to be transmitted, **sendto**() will block unless the socket has been placed in a non-blocking I/O mode. On non-blocking SOCK_STREAM sockets, the number of bytes written may be between 1 and the requested length, depending on buffer availability on both the local and foreign hosts. The **select**() call may be used to determine when it is possible to send more data.

iFlags may be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *iFlags* parameter. The latter is constructed by or-ing any of the following values:

Value	Meaning
MSG_DONTROUTE	Specifies that the data should not be subject to routing.
MSG_OOB	Send out-of-band data (SOCK_STREAM only **Out of band data**)

Return Value

If no error occurs, **sendto()** returns the total number of characters sent. (Note that this may be less than the number indicated by *len*.) Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is returned in errno.

Error Codes

ENETDOWN SOCKETS has detected that the network subsystem has failed.

EACCES The requested address is a broadcast address, but the appropriate flag was not set.

EFAULT The pcBuf or psTo parameters are not part of the user address space, or the psTo argument is too small (less than the sizeof a struct sockaddr).

ENETRESET The connection must be reset because SOCKETS dropped it.

ENOBUFS SOCKETS reports a buffer deadlock.

ENOTCONN The socket is not connected (SOCK_STREAM only).

ENOTSOCK The descriptor is not a socket.

EOPNOTSUPP MSG_OOB was specified, but the socket is not of type SOCK_STREAM.

ESHUTDOWN The socket has been shutdown; it is not possible to sendto() on a socket after shutdown() has been invoked with how set to 1 or 2.

EWOULDBLOCK The socket is marked as non-blocking and the requested operation would block.

EMSGSIZE The socket is of type SOCK_DGRAM, and the datagram is larger than the maximum supported by SOCKETS.

ECONNABORTED The virtual circuit was aborted due to timeout or other failure.

ECONNRESET The virtual circuit was reset by the remote side.

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

EAFNOSUPPORT Addresses in the specified family cannot be used with this socket.

EDESTADDRREQ A destination address is required.

ENETUNREACH The network can't be reached from this host at this time.

See Also

recv(), recvfrom(), socket(), send().

setsockopt

Sets a socket option.

C syntax

int setsockopt (SOCKET *so*, int *level*, int *optname*,

const char * *optval*, int *optlen*);

Parameters

so

A descriptor identifying a socket.

level

The level at which the option is defined; the only supported levels are SOL_SOCKET and IPPROTO_TCP.

optname

The socket option for which the value is to be set.

optval

A pointer to the buffer in which the value for the requested option is supplied.

optlen

The size of the optval buffer.

Remarks

setsockopt() sets the current value for a socket option associated with a socket of any type, in any state. Although options may exist at multiple protocol levels, this specification only defines options that exist at the uppermost "socket" level. Options affect socket operations, such as whether expedited data is received in the normal data stream, whether broadcast messages may be sent on the socket, etc.

There are two types of socket options: Boolean options that enable or disable a feature or behavior, and options which require an integer value or structure. To enable a Boolean option, *optval* points to a nonzero integer. To disable the option *optval* points to an integer equal to zero. *optlen* should be equal to sizeof(int) for Boolean options. For other options, *optval* points to the an integer or structure that contains the desired value for the option, and *optlen* is the length of the integer or structure.

SO_LINGER controls the action taken when unsent data is queued on a socket and a **closesocket()** is performed. See **closesocket()** for a description of the way in which the SO_LINGER settings affect the semantics of **closesocket()**. The application sets the desired behavior by creating a *struct linger* (pointed to by the *optval* argument) with the following elements:

```
struct linger {
    int     l_onoff;
    int     l_linger;
}
```

To enable SO_LINGER, the application should set *l_onoff* to a non-zero value, set *l_linger* to 0 or the desired timeout (in seconds), and call **setsockopt()**. To enable SO_DONTLINGER (i.e. disable SO_LINGER) *l_onoff* should be set to zero and **setsockopt()** should be called.

By default, a socket may not be bound (see **bind()**) to a local address which is already in use. On occasions, however, it may be desirable to "re-use" an address in this way. Since every connection is uniquely identified by the combination of local and remote addresses, there is no problem with having two sockets bound to the same local address as long as the remote addresses are different. To inform SOCKETS that a **bind()** on a socket should not be disallowed because the desired address is already in use by another socket, the application should set the SO_REUSEADDR socket option for the socket before issuing the **bind()**. Note that the option is interpreted only at the time of the **bind()**: it is therefore unnecessary (but harmless) to set the option on a socket which is not to be bound to an existing address, and setting or resetting the option after the **bind()** has no effect on this or any other socket.

An application may request that SOCKETS enable the use of "keep-alive" packets on TCP connections by turning on the SO_KEEPALIVE socket option. If a connection is dropped as the result of "keep-alives" the error code ENETRESET is returned to any calls in progress on the socket, and any subsequent calls will fail with ENOTCONN.

The TCP_NODELAY option disables the Nagle algorithm. The Nagle algorithm is used to reduce the number of small packets sent by a host by buffering unacknowledged send data until a full-size packet can be sent. However, for some applications this algorithm can impede performance, and TCP_NODELAY may be used to turn it off. Application writers should not set TCP_NODELAY unless the impact of doing so is well-understood and desired, since setting TCP_NODELAY can have a significant negative impact of network performance. TCP_NODELAY is the only supported socket option which uses *level* IPPROTO_TCP; all other options use level SOL_SOCKET.

The following options are supported for **setsockopt()**. The Type identifies the type of data addressed by *optval*.

Value	Type	Meaning
SO_BROADCAST	BOOL	Allow transmission of broadcast messages on the socket.
SO_DEBUG	BOOL	Record debugging information.
SO_DONTLINGER	BOOL	Don't block close waiting for unsent data to be sent. Setting this option is equivalent to setting SO_LINGER with *l_onoff* set to zero.
SO_DONTROUTE	BOOL	Don't route: send directly to interface.
SO_KEEPALIVE	BOOL	Send keepalives
SO_LINGER	struct linger *	Linger on close if unsent data is present
SO_OOBINLINE	BOOL	Receive out-of-band data in the normal data stream.
SO_RCVBUF	Int	Specify buffer size for receives
SO_REUSEADDR	BOOL	Allow the socket to be bound to an address which is already in use. (See **bind()**.)
SO_SNDBUF	Int	Specify buffer size for sends.
TCP_NODELAY	BOOL	Disables the Nagle algorithm for send coalescing.

BSD options not supported for **setsockopt()** are:

Value	Type	Meaning
SO_ACCEPTCONN	BOOL	Socket is listening
SO_ERROR	Int	Get error status and clear
SO_RCVLOWAT	Int	Receive low water mark
SO_RCVTIMEO	Int	Receive timeout

SO_SNDLOWAT	Int	Send low water mark
SO_SNDTIMEO	Int	Send timeout
SO_TYPE	Int	Type of the socket
IP_OPTIONS		Set options field in IP header.

Return Value

If no error occurs, **setsockopt()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is returned in errno.

Error Codes

ENETDOWN SOCKETS has detected that the network subsystem has failed.

EFAULT *optval* is not in a valid part of the process address space.

EINVAL *level* is not valid, or the information in *optval* is not valid.

ENETRESET Connection has timed out when SO_KEEPALIVE is set.

ENOPROTOOPT The option is unknown or unsupported. In particular, SO_BROADCAST is not supported on sockets of type SOCK_STREAM, while SO_DONTLINGER, SO_KEEPALIVE, SO_LINGER and SO_OOBINLINE are not supported on sockets of type SOCK_DGRAM.

ENOTCONN Connection has been reset when SO_KEEPALIVE is set.

ENOTSOCK The descriptor is not a socket.

See Also

bind(), getsockopt(), ioctlsocket(), socket().

shutdown

Disables sends and/or receives on a socket.

C syntax

int shutdown (SOCKET *so*, int *how*);

Parameters

so

A descriptor identifying a socket.

how

A flag that describes what types of operation will no longer be allowed.

Remarks

shutdown() is used on all types of sockets to disable reception, transmission, or both.

If *how* is 0, subsequent receives on the socket will be disallowed. This has no effect on the lower protocol layers. For TCP, the TCP window is not changed and incoming data will be accepted (but not acknowledged) until the window is exhausted. For UDP, incoming datagrams are accepted and queued. In no case will an ICMP error packet be generated.

If *how* is 1, subsequent sends are disallowed. For TCP sockets, a FIN will be sent.

Setting *how* to 2 disables both sends and receives as described above.

Note that **shutdown()** does not close the socket, and resources attached to the socket will not be freed until **closesocket()** is invoked.

Comments

shutdown() does not block regardless of the SO_LINGER setting on the socket.

An application should not re-use a socket after it has been shut down.

Return Value

If no error occurs, **shutdown()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is returned in errno**.**

Error Codes

ENETDOWN SOCKETS has detected that the network subsystem has failed.

EINVAL how is not valid.

ENOTCONN The socket is not connected (SOCK_STREAM only).

ENOTSOCK The descriptor is not a socket.

See Also

connect(), socket().

---

socket

Creates a socket.

C syntax

SOCKET socket (int *af*, int *type*, int *protocol*);

Parameters

af

An address family specification. The supported families are PF_INET for IPv4 and PF_INET6 for IPv6.

type

A type specification for the new socket.

protocol

A particular protocol to be used with the socket, or 0 if the caller does not wish to specify a protocol. The supported protocols are IPPROTO_TCP and IPPROTO_UDP.

Remarks

socket() allocates a socket descriptor of the specified address family, data type and protocol, as well as related resources. If a protocol is not specified (i.e. equal to 0), the default for the specified connection type is used.

The following *type* specifications are supported:

Type Explanation

SOCK_STREAM	Provides sequenced, reliable, two-way, connection-based byte streams with an out-of-band data transmission mechanism. Uses TCP for the Internet address family.
SOCK_DGRAM	Supports datagrams, which are connectionless, unreliable buffers of a fixed (typically small) maximum length. Uses UDP for the Internet address family.

Sockets of type SOCK_STREAM are full-duplex byte streams. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a **connect**() call. Once connected, data may be transferred using **send**() and **recv**() calls. When a session has been completed, a **closesocket**() must be performed.

The communications protocols used to implement a SOCK_STREAM ensure that data is not lost or duplicated. If data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, the connection is considered broken and subsequent calls will fail with the error code set to ETIMEDOUT.

SOCK_DGRAM sockets allow sending and receiving of datagrams to and from arbitrary peers using **sendto()** and **recvfrom()**. If such a socket is **connect()**ed to a specific peer, datagrams may be send to that peer **send**() and may be received from (only) this peer using **recv**().

Return Value

If no error occurs, **socket()** returns a descriptor referencing the new socket. Otherwise, a value of INVALID_SOCKET is returned, and a specific error code is returned in errno**.**

Error Codes

ENETDOWN	SOCKETS has detected that the network subsystem has failed.
EAFNOSUPPORT	The specified address family is not supported.
EMFILE	No more file descriptors are available.
ENOBUFS	No buffer space is available. The socket cannot be created.
EPROTONOSUPPORT	The specified protocol is not supported.
EPROTOTYPE	The specified protocol is the wrong type for this socket.
ESOCKTNOSUPPORT	The specified socket type is not supported in this address family.

See Also

accept(), bind(), connect(), getsockname(), getsockopt(), setsockopt(), listen(), recv(), recvfrom(), select(), send(), sendto(), shutdown(), ioctlsocket().

CGI Application API (Server API)

Introduction

The SOCKETS web servers, HTTPD.EXE and HTTPFTPD.EXE, provide both a Spawning Common Gateway Interface (CGI) and an Extension API with the ability to extend the server to create dynamic web pages, perform specialized tasks, etc. One of the extensions provided is a Server Side Includes (SSI) interface using the CGI interface, enabling a user to create web pages using HTML templates with variable names, which is substituted in-time with specific values

The HTTPD Extension CGI works as follows: The extension has to implement one function called the *callback function*. The server has a number of functions that the extension may use, e.g. *HttpSendData*. They are designed to give the extension sufficient control over any http request.

Spawning CGI

An external program, indicated by the requested URL, is spawned. All relevant information is passed as environment variables. The program gets all input (e.g. posted data) from *standard in* and sends all response through *standard out*.

This type of CGI is discouraged in favor of the *Extension API*.

The following CGI environment variables are supported:

CONTENT_TYPE, CONTENT_LENGTH, PATH, COMSPEC and REQUEST_METHOD.

Enough free memory must be available when spawning a CGI program, or no swapping or overlaying will be attempted. Since COMMAND.COM uses all free memory, it follows that no CGI program will be spawned if COMMAND.COM is the current foreground program.

CGI programs must be small and must execute reasonably quickly. While a CGI program is executing, the HTTP server is effectively blocked and cannot service any other requests. No console input or output should be used. A CGI program is invoked by a URL containing a path of /cgi-bin/<cgi-program> where <cgi-program> is the name of an executable program which must be in the HTTP root directory or in the path. Note that the "/cgi-bin/" part is stripped off and does not represent a real directory. <Cgi-program> may be followed by a "?" and a command line. On entry to the CGI program, the environment variables listed above are set up and can be accessed. If a command line is given, it can also be accessed in the normal way.

The CGI program generates a dynamic page by writing to STDOUT. When the CGI program terminates, this output is sent to the remote client (browser). The output can consist of a header and a body part separated by an empty line. If the header contains a “Content-type:” line, the content type will be set to that type and only the body will be sent to the client. Otherwise all the output will be sent to the client using content type “text/plain”. COMMAND.COM can be invoked as a CGI program to perform simple DOS functions e.g. directory listings. The following example performs a directory listing:

http://www.embedded-server.com/cgi-bin/command?/cdir

The next one performs a wide directory listing using a wild-card specification:

http://www.embedded-server.com/cgi-bin/command?/cdir%20*.htm%20/w

Note the use of %20 to specify a space character.

Refer to the INDEX.HTM web page for an example of various ways of calling CGI programs. The NUM.EXE program with source code NUM.C, demonstrates the use of a header and body part building a simple “page visited” web page:

```
printf("Content-type: text/html\n\n”
   ”<html>\n<h1>\nThis page has been visited %d times\n</h1>\n",
   number);
printf("<P><P><A HREF=\"/index.htm\">Back</A>.</html>\n");
```

Forms programming can be performed using either the GET or POST methods. When GET is used, form data is copied to the command line and is limited to 128 characters including the URL part. When the POST method is used, the command line is also built. In addition, form data are available from STDIN and is limited by disk space only. See the forms programming example consisting of FORM.HTM, FORM.EXE and FORM.C for examples of using both the GET and POST methods.

So that you may fully understand CGI programming, this detailed explanation of the server operation is provided.

Whenever HTTPD receives a URL containing "/cgi-bin/", it interprets the rest of the URL as a DOS program to spawn and run to completion. The full path parsed from the URL is used, implying that the program should be in physical directory called "/cgi-bin/" or a subdirectory thereof. E.g. "program.exe" should be in "%HTTP_DIR%\cgi-bin\" if the request is "GET /cgi-bin/program.exe".

While this "CGI program" is executing, the server can accept new server connections, but will not respond to them before the CGI program terminates. The CGI program can be any DOS program that is small enough to fit into available memory. Since HTTPD is blocked while the CGI program executes, user interaction should not be used and the CGI program should complete in a reasonable time.

Operation on receiving a CGI URL:

If the CGI program name is followed by a "?", the rest of the line is sent as a command line to the CGI program after converting all %n combinations.

If a "Content-Type" header is encountered, the CONTENT_TYPE environment variable is set to the given value and if a "Content-Length" header is encountered, the CONTENT_LENGTH environment variable is set to the given value. The PATH and COMSPEC environment variables are copied to the new environment and the REQUEST_METHOD environment variable is set to either GET or POST.

If the POST method is used, the rest of the HTTP message is copied to a temporary file that is then re-directed to stdin. The stdout stream is redirected to another temporary file. After completion of the request, the temporary files are deleted. They will be created in the %HTTPTMP% directory.

The CGI program is now invoked. This program can check the environment variables, access the command line and in the case of a POST, read from stdin. All output that should be passed back to the HTTP client (Browser) is written to stdout. A single header line followed by an empty line, containing "Content-type: *content_type"* may be pre-pended to the data. This line will be used to set the content-type of the data being sent back. If such a header is not found, the content type will be set to "text/plain".

Overview of the Extension API

The SOCKETS HTTP servers (HTTPD/HTTPFTPD) provide a facility to call functions in other modules which may be TSR or transient programs. These functions are referred to as "HTTPD extensions". HTTPD or HTTPFTPD must be loaded as a TSR using the /r switch. It provides an API via software Interrupt 63Hex. The API can be located by searching for a signature containing SockHTTPD starting 10 bytes before the interrupt entry point and terminated by a 0 byte.

A **CGI adapter** is provided that simplifies the communication with the server. It is located in a file called CGIADAP.C. The adapter finds the signature and provides a C interface. It also intercepts the callback function and performs a stack and context switch, which makes implementing an extension much easier.

An HTTPD extension registers interest in a specific URL by calling the **HttpRegister()** API specifying a "path". Note that this path has nothing to do with an actual file path on the server and will override any real path that may be used for serving static pages. The **HttpRegister()** function also specifies a Callback function to be called when the actual request is received by HTTPD, a DWORD User ID to be used in callbacks and whether requests should be allowed to overlap, i.e. a new request can be received while still servicing a previous request or requests.

The Callback function will be called when a request for the registered path is received and as many times afterwards as is necessary to complete the request. It is called with a parameter structure specifying the reason for the request, the User ID, an HTTPD handle and values specific to the reason for the callback, e.g. a pointer to the command line on the initial callback. Other reasons for calling the Callback function are to notify of new received data, connection closure by the peer, readiness to accept more data and connection errors. The callback must return a value to indicate that it is still busy handling the request, has completed the request or wants to abort the request with an error. The HTTPD handle will be constant and unique from the first callback to the completion of the request.

While in the Callback function, data can be read from the peer or sent to the peer and a file can be submitted to be sent to the peer.

Note: Extensions are responsible for sending all HTTP header fields to clients.

The following extensions have been developed for functional and demonstrational purposes.

SSI Interface

If you want to display the current date and time, or a certain CGI environment variable in your otherwise static document, you can go through the trouble of writing a CGI program that outputs this small amount of virtual data. Or better yet, you can use a powerful feature called Server Side Includes (or SSI).

Server Side Includes are directives which you can place into your HTML documents to output such data as environment variables and file statistics.

A simple yet powerful interface is provided to perform Server Side Includes (SSI) tasks. A user only has to implement one predefined function and make use of only four API functions to unlock the power of SSI.

The working of the interface is described at the top of the header file *ssi.h*.

To use, include *ssicgi.c* in your project and include *ssi.h* in your source files. Take a look at *ssi.c* for a simple example.

WebDOS

Introduction

WebDOS is a system allowing a Web browser to provide a user interface to an embedded system. It consists of the WebDosCommander "shell", specific Web-based utilities and a framework to implement web-based applications called WebForms. It is based on Sockets and the Sockets Web Server.

WebDOS Commander presents a "Commander" like interface allowing a user to navigate the directory, copy, view, delete and execute files and create and delete folders (directories).

WebDOS applications can be either web-based or simply utility type programs taking command line parameters and providing normal text output. Web-based applications should be written as small TSR modules which may be invoked by WebDOS Commander and terminated by a user action from the Browser. From the user's viewpoint, WebDOS appears as a multi-tasking, Windows based system as a result of the event-driven nature of the implementation and the use of a Browser.

WebForms

WebForms is a simple framework to assist in writing web-based applications. It is essentially a set of API functions providing an advanced "Server Side Include" functionality. It also includes code to spawn and terminate web oriented TSRs.

Coding for WebForms consists of coding HTML pages using special format comments to retrieve dynamic information. The dynamic information is provided by a user-coded module in C. The HTML pages normally include HTML forms. User input is normally posted back to the user-coded module via WebForms using the normal HTML input controls or HTML links. A WebForms coder needs to have knowledge of HTML and C programming while knowledge of a client-side scripting language like JavaScript is recommended.

WebDosCommander has been implemented using WebForms as the basis.

Writing WebForms programs.

Web-aware applications are normally written as event driven TSR programs. This makes it possible to run an arbitrary mix of them at the same time, effectively making the WebDOS system a windows-based multi-tasking system from the user's perspective. One of the main problems of TSR programming i.e. the issue of DOS re-entrancy, is taken care of by HTTPD.EXE or HTTPFTPD.EXE, whichever is used. To simplify the TSR coding even more UNLOADSTR.C is provided to unload the TSR.

WEBFORM.H defines structures, constants and function prototypes used for writing the WebForms application.

WEBFORM.C makes use of CGIADAP.C. The key functions are **InitForm()** and **FreeForm()**. **InitForm()** is used to define a URL and either a table template or a file template of the HTML code to be generated when the URL is requested by a browser. It also defines entry points to be called when dynamic data must be supplied by the user-written C program and when input is

received from the browser. **FreeForm()** is used to de-register the URL and free memory allocated by **InitForm()**. **SetForm()** is used to switch to a new table template and **SetFile()** to switch to a new file template. More than one form may be initialized by calling **InitForm()** more than once. This will normally be done when using frames with dynamic data.

A file template is a file containing HTML code where HTML comments of a specific structure are replaced by strings supplied by the user-written C program at run-time. A comment like

<!--#***type=id ...*** --> is replaced by one or more strings as defined by ***type*** and ***id***:

<!--#txt=***id***> is replaced by a single text string identified by ***id***.

<!--#txr=***id1*** txr=***id2*** ... txr=***idn*** > is replaced by any number of text strings identified by ***id1 id2 ... idn***.

<!--#ift=***id text***> is replaced by ***text*** when the return form (*pGetData)(id) is positive or by nothing if the return is negative. It is in order for ***text*** to contain matched pairs of sharp backets (<...>).

The ***id*** is a two character alphanumeric string which will be passed in the WORD wId parameter to the C function specified in the **InitForm()** function as the **(*pGetData)()** function. The identified string is copied to the char *pszData buffer (maximum 200 bytes). The return from **(*pGetData)()** specifies the length of the string or –1 if the last string of a series of strings has been returned e.g. when replacing <!--#txr=id>. As shown above, the closing -- of the special comment may be omitted.

Please refer to WTCP.C and WTCPFORM.HTM for examples to code WebDOS programs.

WTCP.EXE together with WTCPFORM.HTM displays a list of the TCP connections on the server.

SHOWLOG.EXE together with ShowLogF.htm is an example to display the log file generated by FTPHTTPD.EXE or HTTPD.EXE on a browser in reverse order. Run HTTPFTPD with logging enabled from the %HTTP_DIR% directory:

```
httpftpd /l=netacces.txt /r
```

Run WebDos Commander:

```
webdosc
```

Now point the browser at the server and select WebDos Commander. Scroll to "ShowLog.exe" and double-click. This will launch the SHOWLOG.EXE TSR and open a new window to show the results. Resize the window for a good view. The Refresh Time and Maximum number of entries to be displayed can be changed. The Stop button will stop the refresh and the Exit button will cause SHOWLOG.EXE to be unloaded on the server and the window to be closed.

ShowLog can be run on the server instead of WebDosC and accessed by the URL:

http://<*Server*>/showlog.htm

WebDOS Commander

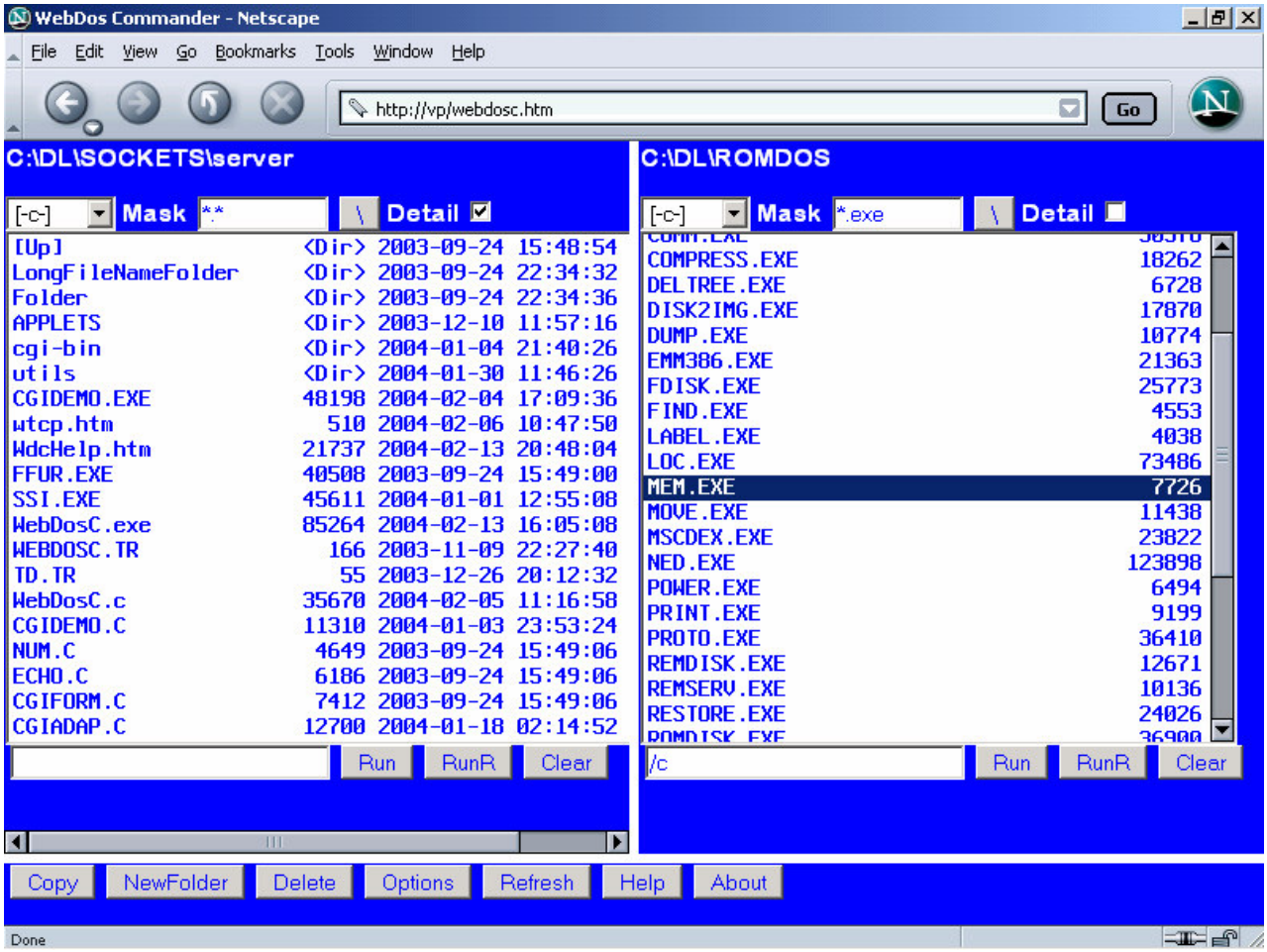
Documentation for WebDOS Commander is contained in WDCHELP.HTM with associated picture files and can be accessed by clicking the **Help** button in WebDOS Commander. WebDOS

commander consists of WEBDOSC.EXE and a number of HTM, JPEG and GIF files. It is normally run as a transient program with no command-line parameters, but can also be run as a TSR (/r) or a transient program spawning COMMAND.COM (/t).

While running as a normal transient program, the keyboard is active. Press the space bar for a list of valid options which is primarily used to give debugging information. Pressing 'd' will invoke COMMAND.COM. While COMMAND.COM is running, trying to use WebDOS Commander to execute any programs will fail with a memory availability error.

Only one non-TSR program can be executed at a time from WebDOS Commander. Attempting to execute any program while running a non-TSR program, will fail with a "Waiting for previous execute to complete" error.

Screenshot of WebDosCommander



Other Extension API Examples

A number of examples are included to demonstrate the usage of both the Spawning and Extension API. Source code is included, as well as a make file and a BorlandC 5 IDE file.

Put all *.htm* and *.exe* files in the %HTTP_DIR% (default is \dl\sockets\server) directory and start *HTTPD, HTTPFTPD or SUPERD*. The CGI programs may be pre-loaded or WebDos Commander may be loaded and used to invoke the CGI programs. Examples may be accessed through *index.htm*.

The only Spawning CGI example is *form.exe* and it is accessed through *form.htm*. Please note again that COMMAND.COM must not be the currently executing program on the server as it uses all available free memory. WEBDOSC.EXE is a good program to have executing on the server.

The first four examples may operate in one of two modes:

As a TSR (resident) program: this is the default behavior. Unloading of the TSR can be performed by the user from the remote browser session or by unloading the server. De-registration is possible by loading the program again. This routine may be repeated.

As a transient program: use the '/t' command line switch to activate. This option will immediately spawn 'command.com'. From this prompt other cgi programs may be loaded. The program exits when 'command.com' is exited by typing 'exit' at the prompt.

These programs are:

CGIDEMO

This program contains three Extension API functions:

1. A simple program that accepts data from a user and echoes it back nicely formatted. Access *echoform.htm* from the browser.

2. A page visit counter. Access *num.htm* from the browser.

3. Does the same as the *form.exe* spawn example. Get *caform.htm* from the browser.

SSI

A simple SSI implementation that demonstrates the SSI interfaces. *Template.htm* is filled by some variables. Get *ssi.htm* from the browser.

FFUR (Form-base File Upload Receiver)

It handles the upload of a file using POST commands. Access *ffur.htm.*

HTTPD Function Reference

CGIADAP.C is an interface program a user may utilize to implement external extension CGI programs. This interface performs stack and context switches, and provides ordinary C functions to access the http server (*HTTPD.exe*).

The header file to use is CGIADAP.H.

The API may be used without using CGIADAP by making low level calls which are detailed below. In this case the user must perform the required stack and context switches if required.

HttpRegister

The **HttpRegister()** function registers an interest in a URL, providing a callback function. The callback is guaranteed to only be called when DOS can be called. The DOS critical handler will be disabled and all critical errors will result in an access error without any user intervention. Since the callback happens at interrupt time, it should execute for as short a time as possible. After a done or error return, no further callbacks will be generated for the current request.

Only one callback will be active at any time. Calling an API function while executing the callback function will not result in another callback before the current callback has returned.

C syntax

int **HttpRegister**(far char *pf*szPath*,

int (far **pfCallback*)(HTTP_PARAMS far **pfsHttpParams*),

int *iFlags*, DWORD *dwUserID*);

Options

pfszPath

far pointer to the string identifying a URL. It should be an exact match of the abs_path part of the URI minus the leading '/'. For instance, If you want to capture all http://myserver.com/cgi-bin/getpage.exe, you should register 'cgi-bin/getpage.exe'.

pfCallback

Address of callback function.

Return values when returning from callback:

RET_OK	not done, give me more upcalls
RET_DONE	done, no more upcalls please
RET_ERR	done, error

pfsHttpParams

Far pointer to HTTP parameter block.

pfsHttpParams->iReason

Reason for callback:

R_NEWREQ -	New HTTP request. pszCommandLine points to the command line passed in the URL. The number contained in iValue specifies the HTTP operation; RQ_GET for GET and RQ_POST for POST.
R_INDATA -	Input data available, iValue contains count.
R_OUTDATA -	Can send output data, iValue contains count.
R_ENDDATA -	Peer closed connection i.e. "end of input data"

R_CLOSED - Connection closed.

pfsHttpParams->iHandle

HTTPD handle, used in subsequent API calls for this request. The user should not modify it.

See HTAPIC.H for the other definitions

iFlags

F_OVERLAP - Overlapped request (1), non-overlapped request (0).

All other bits are reserved.

dwUserID

Value passed to HttpRegister(); this value is for use by the extension, HTTPD does not modify it.

Return value

0: OK

< 0: One of the error messages (SEE HTAPIC.H)

Low level calling parameters

AH APIF_REGISTER (0)

DS:SI pfszPath

ES:DI pfCallback

BX iFlags

CX:DX dwUserID

Low level return parameters

Return code in AX.

Note that the stack and the data segment on entry will be that of HTTPD. Depending on the memory model used for the extension and the amount of stack space required, it may be required to switch stacks during the callback.

HttpDeRegister

The **HttpDeRegister()** function removes the interest in a URL. After this call no more callbacks will be generated for this URL. Any requests in progress will be terminated with an error to the peer. This function must be called for all registrations made by a program before terminating that program; otherwise the system will inevitably crash on any subsequent request.

C syntax

int **HttpDeRegister**(char far **pfszPath*);

Options

pfszPath

Far pointer to URL to de-register.

Return value

0: OK

< 0: One of the error messages (SEE HTAPIC.H)

Low level calling parameters

AH APIF_DEREGISTER (1)

DS:SI pfszPath

Low level return parameters

Return code in AX.

HttpGetData

The **HttpGetData()** function can be called when a POST operation has been indicated by the callback to get data sent to the server by the client. If more data is expected and the extension is busy executing the callback function, a 0 return should be made from the callback indicating it is still busy and getting more data should be attempted at the next callback.

return: >= 0 - ok, bytes received

< 0: One of the error messages (see htapic.h)

C syntax

int HttpGetData(int *iHandle*, char far **pfcBuf*, int *iCount*);

Options

iHandle

Handle passed in pfsHttpParams.

pfcBuf

Far pointer to buffer to receive data.

iCount

Length of buffer.

Return value

>=0: OK, number of bytes received.

< 0: One of the error messages (SEE HTAPIC.H)

Low level calling parameters

AH APIF_GETDATA (2)

BX iHandle

DS:SI pfcBuf

CX iCount

Low level return parameters

Return code in AX.

HttpSendData

The **HttpSendData()** function is used to send data to the client.

If the return indicates that less than the requested number of bytes has been sent and the extension is busy executing the callback function, a 0 return should be made from the callback indicating it is still busy. Then an attempt to send more data should be made at the next callback.

All the required data should be sent to the client before an **HttpSubmitFile()** function is used. After **HttpSubmitFile()**, **HttpSendData()** should not be called again.

C syntax

int **HttpSendData**(int *iHandle*, char far **pfcBuf*, int *iCount*);

Options

iHandle

Handle passed in pfsHttpParams.

pfcBuf

Far pointer to buffer with data to send.

iCount

Length of buffer.

Return value

>= 0: number of bytes actually sent

< 0: One of the error messages (SEE HTAPIC.H)

Low level calling parameters

AH APIF_SENDDATA (3)

BX iHandle

DS:SI pcBuf

CX iCount

Low level return parameters

Return code in AX.

HttpSubmitFile

The **HttpSubmitFile()** function is used to submit a file to be sent to the client in response to a request. The file will be logically appended to any data already sent using **HttpSendData()**. The file should not be exclusively opened when it is submitted. After it is transmitted, transmit upcalls will be issued normally. This gives the user the ability to send any number of files on the connection with arbitrary data in between.

C syntax

int HttpSubmitFile(int *iHandle*, char far **pfszFileName*);

Options

iHandle

Handle passed in pfsHttpParams.

pfszFileName

Far pointer to name of file to submit.

Return value

0: OK

< 0: One of the error messages (SEE HTAPIC.H)

Low level calling parameters

AH APIF_SENDFILE (4)

BX iHandle

DS:SI pfszFileName

HttpGetStatus

The **HttpGetStatus()** function gets the number of connections to the server. It must also be used as a polling function when the server is running in passive mode to dequeue and handle pending requests.

C syntax

int **HttpGetStatus**(void);

Return value

>=0: Number of connections to server.

< 0: One of the error messages (SEE HTAPIC.H)

Low level calling parameters

AH APIF_GETSTATUS(6)

Low level return parameters

Return code in AX.

HttpGetVersion

The **HttpGetVersion()** function gets the version of the running HTTP server.

C syntax

int **HttpDeRegister**(void);

Return value

>=0: Version number.

< 0: One of the error messages (SEE HTAPIC.H)

Low level calling parameters

AH APIF_GETVERSION (5)

Low level return parameters

Return code in AX.

GetStackPointer/GetStackSegment

The **GetStackPointer()/GetStackSegment()** functions get the current Stack Pointer/Segment.

C syntax

int **SetStackPointer** (void);

int **SetStackSegment** (void);

Return value

Current value of Stack Pointer/Segment.

SetStackPointer/SetStackSegment

The **SetStackPointer()/SetStackSegment()** functions set the Stack Pointer/Segment.

The stack pointer for callbacks is by default set to _SP - 1000, the first time the HTTP API is called. If you would need space on the stack, or for some reason want to make it tighter, set the stack pointer for callbacks manually. Be careful not to overwrite used memory.

C syntax

void **SetStackPointer** (int iPointer);

void **SetStackSegment** (int iSegment);

Options

iPointer

Value to set Stack Pointer to.

Return value

None

Constants and Definitions used by CGI API

Refer to HTAPIC.H.

SSI Definitions and functions

Refer to SSI.H.

Other APIs

FTP API

FTPAPI provides both a server and client FTP API. This API is loaded using the FTPAPI.EXE TSR program and provides an assembler level interface at Interrupt 62Hex. It should be called directly from your application. The documentation for this interface is found in FTPAPI.H. A

complete “server and multiple client” sample program is provided as FTPTEST.C and FTPTEST.EXE. It also demonstrates using C functions to call the API.

NETBIOS

The industry-standard NETBIOS API is loaded by using the NETBIOS.COM TSR program. This API is widely documented and the protocols used are those specified by RFC1001 and RFC1002. The major use of this API is to run existing NETBIOS applications like the file redirector and file server provided by the freeware PowerLAN software which provide file sharing with other systems using SMB or IFS. Examples of that are Microsoft Lanman, Microsoft Windows and Unix/Linux SAMBA.

SOCKETS Proprietary API

The SOCKETS Proprietary API is modelled very closely to the internal structure of the SOCKETS kernel, which hides few details from the programmer. As a result, it is more difficult to work with, and should be used only when its extended features and lowered memory footprint are required. The documentation is only provided inside the API.H source file. The API uses Interrupt 7F Hex by default.

Most of the application programs supplied with SOCKETS use this API. An example is XPING.EXE which is also supplied in source form as XPING.C.

Chapter 4, Tutorials

Building ROM-DOS

Building ROM-DOS is accomplished with single utility named BUILD. BUILD allows you to specify such parameters as which drive ROM-DOS should boot from, where ROM-DOS will run (in ROM or RAM), and, if in ROM, what sort of file(s) your PROM programmer requires. It also allows you to specify whether to BUILD the 7.1 or the 6.22 version of ROM-DOS.

BUILD operates interactively, prompting for information and option selections. BUILD requires an assembler and a linker – we recommend the Borland 5.2 tools provided with the Datalight SDTK. BUILD also requires a locator and a specialized program that compresses the ROM-DOS data. Datalight provides these programs, named LOC.EXE and COMPRESS.EXE, with the Software Development Tool Kit. These programs must be available in the current directory or in the specified path.

BUILD performs the following operations:

- Assembles the SYSGEN (ROM-DOS configuration) file
- Links the ROM-DOS kernel
- Compresses ROM-DOS data
- Locates the ROM-DOS kernel

BUILD creates only the ROM-DOS kernel. In the case of a ROM-DOS kernel that is bootable from a floppy or hard disk, the file created is ROM-DOS.SYS. In the case of a ROM-DOS kernel that is bootable from a ROM, the file created is ROM-DOS.IMG or ROM-DOS.HEX.

Most programs, such as the command interpreter and DOS utilities (FORMAT, SYS, and so on), never need to be configured; they are standard across all systems.

Note: Under some circumstances BUILD may not be flexible enough to meet the special needs of your system. For instance, ROM-DOS in ROM normally gains control via a BIOS extension, and it may be necessary for ROM-DOS to receive control via a direct jump rather than using a BIOS extension. See Chapter 8 for more information on these custom changes.

BUILD Command Line Options

Ordinarily, BUILD will be run without any command-line options. It will then determine the appropriate display colors and find the assembler and linker. These command-line options are provided to correct certain error conditions.

Option	Description
	Causes BUILD to locate ROM-DOS without assembling or linking.
	Causes BUILD to use monochrome. Non-color displays that appear to be color displays to BUILD, such as LCD displays, may not be readable in full color.
	Causes BUILD to pause after running each sub-program. This option allows you to observe what command-lines BUILD is passing to the assembler, linker and so on.
	Causes BUILD to display in TTY mode rather than graphics. This option is necessary for incompatible monitor types.

BUILD can rerun the last session using a configuration file. Each time BUILD runs, it saves a list of your keystrokes in a file named BUILD.CFG. This file can be used, through a standard DOS pipe into BUILD, to repeat the last session. For example:

```
C:\>BUILD < BUILD.CFG
```

If a number of standard sessions are planned, copy the file BUILD.CFG to some other name. Then redirect that filename into BUILD any number of times. BUILD also creates a file named BUILD.TXT. This file contains a complete list of the questions and the answers you selected during the last BUILD session and is the same information as on BUILD's final confirmation screen. BUILD.TXT can be referenced when calling technical support or saved with your project for future reference.

The third output file from BUILD.EXE is BUILD.BAT. BUILD.BAT contains a complete set of instructions for assembling, linking, compressing, and locating the version of the ROM-DOS kernel set up in the previous run of BUILD. Executing BUILD.BAT generates a copy of the previous ROM-DOS kernel without running the BUILD program. BUILD.BAT relies on the existence of two other files, ROM-DOS.LNK (linking command line) and ROM-DOS.LOC (location configuration file). Both files are generated during the BUILD session.

Note: To run BUILD.BAT, you must specify the .BAT extension, otherwise the .EXE extension is assumed and BUILD.EXE runs.

Datalight recommends saving a copy of BUILD.BAT, ROM-DOS.LNK, ROM-DOS.LOC and BUILD.TXT under different names or in a separate directory when you successfully create a working ROM-DOS kernel. This ensures that you can always re-create the same working ROM-DOS kernel configured for your exact needs.

Note: Each revision of BUILD may change: do not use old configuration files on a new BUILD.

If you want to change the default colors, specify the new colors in a text file named BUILD.COL. The colors must be listed as four comma-separated integers, on the first line of the file. The numbers represent the background, window, error, and question colors, using the standard color mapping. For example, to set a gray background with white text, a blue text window with white text, a red error window with white text, and a blue question prompt with yellow text, enter:

```
C:\>COPY CON BUILD.COL
127, 31, 79, 30 <Ctrl+Z>
```

Before Running BUILD

Before you run BUILD, you will need to be prepared to make several decisions based on your hardware, your application program needs, and your PROM programmer if needed. ROM-DOS 7.1 provides many options for configuring different levels of support.

- **Will you need DOS 7.1 compatibility or DOS 6.22 compatibility?** DOS 7.1 gives you the ability to work with FAT32 format disk drives. DOS 6.22 can only work with FAT16 and FAT12 format drives. FAT16 drives have a maximum size of 8-Gig divided into partitions with a maximum size of 2-Gig each. FAT32 drives have a theoretical limit of 2 Terabytes. Remember to consider all of the systems that may be linked in any way to your target system. If not all of them can access a FAT32 drive, you many want to consider a FAT16 drive.
- **Will you need Long Filename support?** ROM-DOS's COMMAND.COM and kernel can be configured to allow the use of Long Filenames. The DOS kernel provides support for Long Filenames through standard Int21h functions. The command processor provides Long Filename support and recognition for its internal commands such as DIR and COPY. Please refer to the User's Guide and Chapter 8 for more information on Long File Names.
- **Will ROM-DOS boot from a floppy or hard drive or bootable flash drive?** If not, will the ROM bootable version of ROM-DOS need to copy itself to RAM at run time for speed reasons?
- **What drives will you have available on your system?** Some common choices are floppy, hard-drive, ROM, RAM, flash, and custom memory disks.
- **For DOS 6.22, what level of CONFIG.SYS processing support is needed?** What drive (letter or type) will you store the CONFIG.SYS file on?
- **Does your PROM programmer require binary or Intel hex files?**

BUILD provides details for each question that is asked. The final screen displays a summary of the information and the size of the ROM image file or disk system files that have been created.

BUILD Sample Sessions

BUILD allows you to create both a floppy/hard disk bootable version of ROM-DOS and a ROM bootable version. You must have the assembler and linker in your path (Borland's TASM/TLINK combination) for BUILD to complete its process. (These tools are available in the Developer's Toolkit*).* If BUILD does not find an available assembler/linker, it warns you and gives you an option to proceed anyway or quit the BUILD process. Several examples are shown below. The output from BUILD is shown in block letters. The user-entered responses are shown in bold. You can press Esc to exit BUILD at any time.

Example 1: Creating a Bootable Version of ROM-DOS on a Floppy Disk

This example creates a ROM-DOS on a floppy disk that can be used to boot your system. To begin, insert a formatted floppy disk in drive A. It doesn't matter if there are files on the disk, as long as there is enough free space for ROM-DOS and its command interpreter (about 80KB). If you do not have a formatted floppy disk, use Datalight's FORMAT.COM program to format the floppy.

To make the disk bootable, you will need to have the file ROM-DOS.SYS. The Software Development Kit provides a ready-made copy of this file. If you need to re-create the file, this can easily be done using the Build utility.

```
C:\DL\ROM-DOS> BUILD
Do you wish to Quick-Build or Custom-Build ROM-DOS (Q/C): Q
Would you like DOS 7.1 compatibility? Y
Would you like to enable LFN support? N
Will ROM-DOS boot from Floppy/Hard disk? Y
```

Change to the ROMDOS directory on your hard disk (or whichever directory you chose to create during the INSTALL) and run the Datalight SYS program. Please note, SYS and other utilities such as FORMT are located in the UTILS subdirectory in original ROM-DOS 6.22 installations.

```
C:\DL\ROM-DOS> SYS A:
```

The SYS utility creates a bootable disk, creates a special boot sector, and copies the ROM-DOS kernel files and the command interpreter (COMMAND.COM) onto the disk. SYS uses the single file ROM-DOS.SYS, produced by BUILD, to generate the two system files, IBMBIO.COM and IBMDOS.COM. These two files are placed as hidden files on the bootable disk. To verify the existence of these files, you can use the DIR command with the system file attribute options as follows:

```
C:\DL\ROM-DOS> DIR A: /AS
```

The SYS program requires that both ROM-DOS.SYS and COMMAND.COM are available in the current directory, or that Datalight's IBMBIO.COM and IBMDOS.COM are in the root directory of a currently booted floppy or hard disk. If SYS cannot find these files, it prompts you for a path for their location.

You can also use the FORMAT utility to both format a disk and add the system onto it as follows:

```
C:\DL\ROM-DOS> FORMAT A: /S
```

Example 2: Creating a Version of ROM-DOS in a ROM

This example uses a standard PC with a ROM card to produce a version of ROM-DOS (in ROM) that may be used on any desktop PC/AT. In this example, ROM-DOS processes CONFIG.SYS and loads COMMAND.COM from the floppy, but boots and executes from ROM. The system files IBMBIO.COM and IBMDOS.COM are not required to be on the floppy disk.

The BUILD.EXE utility is the tool used to create a ROM version of ROM-DOS and prompts with a number of questions during the custom-build session described below. The Quick-Build is usually more appropriate and much easier to run through, but in this example we are booting from ROM but loading CONFIG.SYS from floppy disk.

The following list shows the output that BUILD provides showing the prompts and the appropriate responses for this example.

```
C:\DL\ROM-DOS> BUILD
Do you wish to Quick-Build or Custom-Build ROM-DOS (Q/C): C
Would you like DOS 7.1 compatibility? N
Would you like to enable LFN support? N
Will ROM-DOS boot from Floppy/Hard disk? N
Copy ROM-DOS to RAM? N
Where shall ROM-DOS data reside [70]: 70
Can a Floppy DOS superscede ROM-DOS in ROM? N
Do you want to include the Floppy/Hard disk Driver? Y
Would you like to enable SuperBoot support? N
Always believe the BPB information? N
Include the Custom Memory Disk Driver? N
Include the built-in ROM-DISK driver in ROM-DOS? N
Read CONFIG.SYS from a specific drive letter? Y
Read CONFIG.SYS from which drive letter: A
What level of CONFIG.SYS processing (None, 3, 5, 6)? 6
Do you want ROM-DOS boot diagnostics? Y
Include the Boot Menu? N
Use Real Time Clock Exclusively? N
Create Binary or Intel HEX file(s) as output (B/H): B
Split the output into Odd byte and Even byte files? N
```

The preceding example assumes you have a ROM board to plug into your desktop PC/AT, an EPROM programmer and a ROM large enough to hold ROM-DOS (approx. 60KB).

Program the ROM-DOS.IMG file into an EPROM, plug it into the ROM board and set the address to D000:0. Plug the board into your desktop PC/AT, place a floppy in drive A: with Datalight's COMMAND.COM on it and apply power. ROM-DOS proceeds to check for a CONFIG.SYS file and COMMAND.COM on drive A: (the floppy). At the DOS prompt, type:

```
A:\> VER /R

Datalight ROM-DOS Version 6.22
Copyright (c) 1989-2001 Datalight, Inc.

Kernel Reports Version        6.22
Kernel Resides in             ROM
Kernel Revision               4.11.1403
Command Revision              4.11.1403
```

The VER command (with the revision option) displays the ROM-DOS version and where it is running (ROM, RAM or high memory area).

If you want to boot from the DOS on your hard disk and bypass ROM-DOS in ROM, it is not necessary to remove the ROM card. Hold down the Alt-key while the system boots and ROM-DOS displays a menu of boot options. (Select Yes to the boot menu option during BUILD to activate this feature.) Choose the menu option to boot DOS from hard disk.

Example 3: Creating a Diskless System with ROM-DOS

This example places ROM-DOS and a ROM disk into ROM on an AT motherboard. The example assumes the AT motherboard has 128KB of ROM space. The ROM-DOS kernel and a ROM disk

are placed in the ROM. The ROM disk contains the files COMMAND.COM, TRANSFER.EXE, VDISK.SYS and CONFIG.SYS. This example creates binary images used for input by the PROM programmer. The image files can be Intel hex files or split files, depending on the needs of your programmer.

ROM-DOS requires about 57KB ROM and the ROM disk another 72KB for a total of 126KB ROM space. These sizes may change as new features are added to the BIOS, ROM-DOS or the command interpreter.

These two files are common to most diskless systems.

- ROM-DOS kernel (ROM-DOS.IMG)
- ROM disk (ROMDISK.IMG)

The file ROM-DOS.IMG is created using the BUILD program as in the previous example. This example uses Quick-Build instead of the Custom-Build to simplify the operation shown below.

```
C:\DL\ROM-DOS> BUILD
Do you wish to Quick-Build or Custom-Build ROM-DOS (Q/C): Q
Would you like DOS 7.1 compatibility? N
Would you like to enable LFN support? N
Will ROM-DOS boot from Floppy/Hard disk? N
Create Binary or Intel HEX file(s) as output (B/H): B
Split the output into Odd byte and Even byte files? N
```

BUILD has now created the file ROM-DOS.IMG. Place this file in your PROM programmer directory. Refer to 'Chapter 10, Programming ROM-DOS into ROM' if you have any difficulty during this stage.

Finally, create the ROM disk. You can do this by placing the previously mentioned files into a directory tree and running the ROMDISK.EXE utility. Use the following DOS commands:

```
C:\DL\ROM-DOS> MKDIR TEMPDIR
C:\DL\ROM-DOS> COPY COMMAND.COM TEMPDIR
C:\DL\ROM-DOS> COPY TRANSFER.EXE TEMPDIR
C:\DL\ROM-DOS> COPY VDISK.SYS TEMPDIR
C:\DL\ROM-DOS> COPY CON TEMPDIR\CONFIG.SYS
DEVICE=VDISK.SYS 64 <Ctrl+Z>
```

Now run the ROMDISK.EXE program as shown to create the ROM disk with the files COMMAND.COM, TRANSFER.EXE, VDISK.SYS, and CONFIG.SYS.

```
C:\DL\ROM-DOS> ROMDISK TEMPDIR
\COMMAND.COM
\TRANSFER.EXE
\VDISK.SYS
\CONFIG.SYS

ROM Disk Image     Volume 'ROM-DISK   '
Built from         C:\DL\ROM-DOS\TEMPDIR\*.*
Placed in          ROM-DISK.IMG

    95232 bytes total ROM disk size
      128 bytes in boot sector
      1152 bytes in 7 FAT sectors
      256 bytes in root directory
```

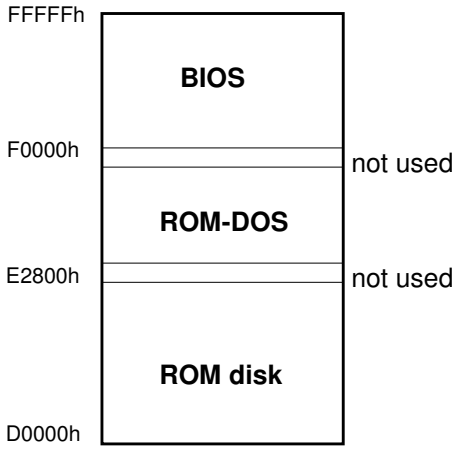
```
        93696 bytes in 4 user file(s)
            0 bytes available on disk

          128 bytes in each of 744 sectors
```

The ROMDISK program creates the file ROM-DISK.IMG. Place this file into the PROM programmer directory along with ROM-DOS.IMG. If you have difficulty creating a ROM DISK, see the section Creating a ROM Disk.' below. Now you need to program these images into one ROM. One way this can be done is to use PROMERGE, which is provided in the SDTK.

```
C:\PROMS> PROMERGE 128K rom-disk.img 0 rom-dos.img -128K
```

Your memory MAP then should look like the following:



Diskless ROM-DOS Memory Map

The ROM-DISK PROM is placed in physical address D000:0 through E000:27FF. The ROM-DOS PROM is located at E000:2800 through E000:FFFF. The system is assumed to contain a BIOS which is located at the standard address, F000:0 through F000:FFFF

Turn on the power to the AT motherboard and the system boots using ROM-DOS.

ROM-DOS is now up and running on a diskless system. This system has a ROM disk as drive A: and a 64KB RAM disk (with the help of VDISK.SYS) as drive B:. The TRANSFER program placed on the ROM disk allows you to copy programs over the console serial port into the RAM disk.

Creating a ROM Disk

A ROM disk, like a fast write-protected floppy disk, contains all of the parts of a standard disk. Each disk consists of a boot sector, a File Allocation Table (FAT), a root directory, and any files that are to be included on the disk. From the point of view of ROM-DOS and any application, the ROM disk appears as a normal disk drive.

The ROM disk image is built using the ROMDISK.EXE utility, which creates ROM disks from a directory tree on your hard disk. The file that the ROMDISK utility outputs is suitable for input to your PROM programmer. The ROMDISK utility can create:

- ROM disks up to 32MB
- ROM disks with directories and subdirectories
- ROM disks containing programs that can execute-in-place (refer to Appendix C)

A ROM disk is typically used in diskless systems to hold applications and/or data. A ROM disk is similar to a RAM disk used under DOS, except that it is read-only and always resides in ROM or linear flash memory.

For the ROM disk to be recognized by ROM-DOS and used as a boot drive, a piece of code called the ROM disk driver must be included within the ROM-DOS kernel. The BUILD program, which creates versions of ROM-DOS specifically for your system, provides the option of including the ROM disk driver. Custom memory disk drives, which recognize and use ROM disk images, can also be built-in to ROM-DOS or loaded by means of CONFIG.SYS.

Running ROMDISK To Create a Disk in ROM

You can run the ROMDISK utility at the command line by entering “ROMDISK” with or without command line options. When run without command line options, ROMDISK displays a summary of the available options. ROMDISK allows you to produce a binary image or an Intel hex file, representing the ROM disk. This file is programmed into ROM to create a ROM disk. The ROM disk has the contents of a standard disk including a boot record, FAT, root directory and data area. The sector size, which defaults to 128 bytes, may be set by entering the sector size on the ROMDISK command line. There is no limit to the number of files that may be placed on a ROM disk other than the above-mentioned limit of 32MB.

Place all of the files to be included on the ROM disk in a directory (the directory may contain subdirectories). The directory you create becomes the root directory on the ROM disk. All subdirectories remain at levels below the root directory.

Use the /S option to transfer subdirectories to the ROM disk. Without the /S option, only the files in the root directory are included. The syntax is similar to XCOPY, with the destination always a file containing the ROM disk. The following example places the contents of the TEMPDIR directory, including subdirectories, in the image file DISK.IMG.

Example:

```
C:\DL\ROM-DOS> ROMDISK TEMPDIR DISK.IMG /s

\COMMAND.COM
\TRANSFER.EXE
\VDISK.SYS
\CONFIG.SYS
\UTILS\FORMAT.COM
\UTILS\SYS.COM

ROM Disk   Volume 'ROM-DISK   '
Built from C:\DL\ROMDOS\TEMPDIR\*.*
Placed in  DISK.IMG

   114944 bytes total ROM disk size
      128 bytes in boot sector
     1408 bytes in 11 FAT sectors
      256 bytes in root directory
      128 bytes in 1 directories
   113024 bytes in 6 user file(s)
        0 bytes available on disk

      128 bytes in each of 898 sectors
```

The file created by ROMDISK.EXE defaults to an image file but can also be an Intel hex file using command line options. The created file must be programmed into ROM and located in the 80x86 memory space. The ROM disk must always start on a paragraph boundary.

ROMDISK Options

The ROMDISK utility has the following command line options to configure the output file. Two of these options are used for the RXE tools. Please refer to Chapter 11 for more information.

Option	Description
/D<Seg>	Defines the RXE Data Segment Name (default is DATA).
/E	Prevents extended records from being placed in the Intel hex output. These records are ordinarily placed in a hex file by default.
/F#	Sets the fill bytes. The default is a fill byte of 0xFF. The number following the /F option is assumed to be in hexadecimal format.
/H[#]	Produces an Intel hex file. An optional number following the /H option specifies the actual address of the start of the ROM disk. The start address is required for EXE files that are to be placed in and executed from ROM. The default address is 0xC000.
/I[#]	Produces an image file. The optional number following the /I option specifies the actual address of the start of the ROM disk. The start address argument is required for EXE files that are to be placed in and executed from ROM. The default address is 0xC000.
/O	Omits the timestamp from the volume label.

Option	Description
/R#	Chooses a hexadecimal interrupt for RXE (default is 90h).
/S	Tells ROMDISK to include all the subdirectories found within the selected source directory in the ROM disk image.
/T	Displays statistics on the ROM disk but does not actually create the image or hex file. This is useful when you need to make sure that the required files fit into the available space.
/V“str”	Sets the volume label to something other than ROMDISK. The volume label string can be up to 11 characters and must be in quotes.
/Z#	Specifies the sector size, in decimal, of the ROM disk. The default sector size is 128 bytes. The only legal values for this option are 128, 256, and 512.

Configuring the ROM Disk Device Driver

The ROM disk image produced by ROMDISK.EXE can be placed in a system in several ways: in conventional memory (under the 1MB boundary), paged into a window in the 80x86 address space, or in extended memory.

The standard ROM disk device driver provided in ROM-DOS supports a ROM disk in conventional memory between the addresses of 40:0 and FFFF:0. This standard ROM disk device driver searches for the start of the disk beginning at a specified segment (usually C000:0). The starting search segment can be specified while building ROM-DOS with BUILD.EXE.

A ROM disk device driver that searches for a ROM disk in paged memory, or extended memory, may be developed by using the code templates provided in the MEMDISK subdirectory. Refer to ‘**Using a Custom Memory Disk**’ on page 167 for more details. If the standard built-in ROM disk driver must be modified, the code can be found in DEVROM.ASM in the DEVSRC subdirectory.

Including Device Drivers

ROM-DOS communicates with hardware through both built-in and installable device drivers. These drivers process all the low-level I/O and hardware-related functions such as setting the system clock, reading from a disk or writing to the display. This processing frees ROM-DOS from the task and, more importantly, from any knowledge of the hardware platform.

This section provides an overview of the device drivers that are built-in to the ROM-DOS kernel, a list of required and optional device drivers, as well as methods for including new or modified device drivers in your ROM-DOS installation. This chapter also assumes that you understand the term “device driver” and are familiar with how to install a device driver by statements in the CONFIG.SYS file.

ROM-DOS Device Drivers

ROM-DOS includes all the device drivers necessary to start a system from ROM, floppy disk, or hard disk. In addition to built-in device drivers, this SDK includes sample device drivers that can be installed from CONFIG.SYS or built-in to the ROM-DOS kernel.

Built-in device drivers are those drivers that have been linked in with ROM-DOS. They are initialized before installed device drivers get initialized, and are generally those devices that will remain standard and constant on your system.

Since ROM-DOS does all of its communication with the hardware through device drivers, a few built-in drivers are mandatory to start a system. These built-in device drivers include the console, the clock, and at least one disk device (either the ROM disk device driver, floppy/hard disk device driver, or a custom disk device driver).

The console is required to display error messages. In systems with no console (or those with a serial port acting as a console), the console driver may be modified to display no output. The clock driver is needed to update the date and time of files as well as to provide the DOS date and time functions.

A disk driver is required in any system. At least one disk must be available to ROM-DOS to find and process CONFIG.SYS and/or load the command interpreter or initial application, whichever applies. ROM-DOS halts if there are no disk devices. The other built-in device drivers, listed below, can be helpful in many systems, but are not required by ROM-DOS.

- Console (CON)
- Clock (CLOCK$)
- Printer (PRN)
- Serial (AUX)
- Com port (COM)
- Null (NUL)
- Floppy/hard disk
- ROM disk

The ROM-DOS SDK includes the source code for a variety of configurable device drivers. These drivers, found in the MEMDISK and DEVSRC directories, can be compiled and installed in CONFIG.SYS or built-in to the ROM-DOS kernel to add functionality to the operating system. No attempt is made in this section to describe the contents of any of these drivers or how they operate.

Writing Device Drivers

Typically, the only type of device driver you need to write for ROM-DOS is an installable driver, the type loaded from CONFIG.SYS. Installable device drivers are the most flexible way to create a driver under ROM-DOS. There is no limitation to the size of the code of the device driver, other than available RAM. Character devices can even override the built-in character devices by using exactly the same name as the built-in counterpart.

There is only one drawback to making a device driver installable from CONFIG.SYS instead of built-in. The driver must either be present on one of the built-in disks or on a disk device that has been previously installed. The device drivers provided in the MEMDISK subdirectory can be used as templates for writing your own custom device drivers.

If you want to write a built-in device driver (one that is linked in with ROM-DOS and not installed through CONFIG.SYS), here are some special considerations:

- The segment nomenclature must agree with ROM-DOS.
- The total ROM-DOS code must be less than 64KB.
- The total ROM-DOS data and stack must be less than 64KB.
- The return address upon initialization is ignored. Built-in devices cannot allocate memory.
- Multiple devices in one file are allowed, but require special treatment.
- The code cannot modify itself in any way.
- ROM-DOS supported math functions need to be used instead of compiler math function. Refer to '**Using a Custom Memory Disk**.'

One reason for adding a new built-in device into ROM-DOS is to provide a new type of disk device from which CONFIG.SYS is processed or the starting application is loaded. ROM-DOS already has built-in floppy, hard, and ROM disk drivers.

The command interpreter COMMAND.COM can typically be loaded from a disk device installed during CONFIG.SYS processing, so COMMAND.COM does not need to reside on a built-in disk.

Note: Device driver names should conform to standard DOS 6.22 8.3 file naming conventions even if the DOS kernel supports long filenames.

Adding New Device Drivers

Installable device drivers are included in your ROM-DOS installation by using a DEVICE= statement in the file CONFIG.SYS. For example:

```
DEVICE=VDISK.SYS 1024 /e
```

The standard built-in device drivers are located in the object library file named ROMDOS.LIB. When ROM-DOS is built, the linker (TLINK) loads each device specified in SYSGEN.ASM from the ROMDOS.LIB file. If you have created your own built-in driver, you can either add the driver to the ROM-DOS library or make a new library named USER.LIB. Datalight recommends adding new or replacement drivers into USER.LIB to make sure the integrity of the original ROM-DOS library file is not compromised.

If the driver you created is intended to replace a standard built-in driver, ROM-DOS uses the driver in the USER.LIB file instead of the driver of the same name in the file ROM-DOS.LIB. USER.LIB drivers always take precedence over ROM-DOS.LIB drivers when there are two drivers with the same name.

Note: If you are using Datalight's MEMDISK.ASM and a client code module, no changes to the SYSGEN.ASM file need to be made. Follow the instructions outlined in '**Using a Custom Memory Disk**' for creating a built-in device and using the BUILD.EXE utility. For all other custom devices, use the following instructions.

The new device driver source files must be compiled or assembled into object files. This can be done with Borland language tools and with reference to the compiler manuals.

Example:

```
Tasm /Mx devprn.asm
```

Place the resulting object files into the ROMDOS.LIB or USER.LIB file using a library maintenance utility (TLIB). The next time ROM-DOS.SYS is linked, the new device drivers will be included in that version of ROM-DOS.SYS

Using Borland's library maintenance utility (TLIB), the following command replaces the ROM disk device driver object file (.OBJ) in the original ROMDOS.LIB file. It also produces a file named ROMDOS.LST that lists the object files in the library and all public labels.

```
C:\>TLIB ROMDOS.LIB –+devrom.obj, romdos.lst
```

To add DEVROM.OBJ to the USER.LIB library file instead, type:

```
C:\>TLIB USER.LIB +devrom.obj,  user.lst
```

If the file USER.LIB did not previously exist, TLIB creates it.

Note: The version of TLIB provided in the Datalight SDTK requires HIMEM.SYS to be loaded when it is run from a DOS platform.

Adding Device Drivers to SYSGEN

Once a built-in device driver has been compiled (or assembled) and added to one of the library files (ROMDOS.LIB or USER.LIB), update SYSGEN.ASM to include the device driver. If the built-in driver is a replacement for a standard built-in driver (such as _comx for COM1-COM4), no changes need to be made to SYSGEN.ASM. If the built-in driver is new, such as a special built-in disk driver, then SYSGEN.ASM must be informed. To do this, locate the section "A NULL Terminated Array of Built-in Devices" in SYSGEN that lists all built-in drivers. Add the following lines to the appropriate group of definitions:

```
extrn _newdiskx:byte

dw OFFSET _DEVDATA:_newdiskx
```

Substitute the name of your driver for _newdiskx. These lines inform the linker (TLINK) that a label by the name of _newdiskx should be linked into the ROM-DOS program. This label is found in either the ROM-DOS or USER library (ROMDOS.LIB or USER.LIB).

Using a Custom Memory Disk

ROM and RAM disks are the basis of diskless systems. and the possible configurations for such disks are as varied as the systems using them. A RAM disk may be implemented as battery-backed static RAM, a ROM disk as paged EPROMs, or a disk could be created using flash memory, which can later be updated in the field.

A ROM disk is necessarily built into a diskless system, enabling ROM-DOS to read CONFIG.SYS and/or load the first program. A built-in RAM disk may also be required in some cases. Installable devices are desirable for some systems due to the advantage of command line options over built-in device drivers.

ROM-DOS includes complete source code for several types of memory disks. Look in the MEMDISK subdirectory for examples. The examples include a paged ROM disk, a RAM disk, a ROM disk, and an extended memory disk driver.

The ROM-DOS configurable memory disk allows you to build a custom memory disk device driver by modifying only the initialize, read, and write, functions. This disk can be configured to be either built-in or installable (loaded by CONFIG.SYS). The custom memory disk is made up of two modules, the base module and the client code module. The memory disk base module is named MEMDISK.ASM and handles all of the direct interaction with DOS. The client code modules are named to reflect their function, such as MEMPAGED.C and MEMROM.C.

Creating a Custom-Memory Disk

The process for creating a custom memory disk consists of the following basic steps:

- Review the source code modules of the intended driver in the MEMDISK subdirectory. For example, for a fixed address disk, review MEMFIXED.C; for a paged memory disk, review MEMPAGED.C and SC400PAG.C. (SC400PAG.C is an example paging algorithm set up for an Elan SC400 platform). The header of each module describes the driver type and uses. Make appropriate modifications to the source code based on the commented sections in the beginning of each module. For example, for a paged disk, modify the paging routines to match your hardware paging mechanism; for a fixed address disk, supply the destination address for the memory disk.
- Review the MAKEFILE section appropriate to the driver you are building. Most sections contain two target output files. For example, there are instructions for building MEMPAGED.SYS (the CONFIG.SYS installable version of the driver) and MEMPAGED.LIB (the built-in version of the driver) in the paged memory disk section. The target name is MEMPAGED.LIB: however, the actual output file is named USER.LIB.
- Switch to the MEMDISK subdirectory and run Borland's MAKE utility with the appropriate target name. For example, to create a built-in paged memory disk, run the following:

```
MAKE MEMPAGED.LIB
```

- To create a stand-alone installable version, run:

```
MAKE MEMPAGED.SYS
```

Note: If the driver is prepared as built-in, a new USER.LIB file is created in the ROM-DOS root directory. If you already had other items in a USER.LIB file, you may want to make a copy of the original file. You may also have to use the TLIB command to add individual object modules into the library file if more than one memory disk or other custom code is to be included in the USER.LIB file

The BUILD program handles the remainder of the process. Refer to 'Chapter 4, Building ROM-DOS' for more information on the BUILD program. Part of the BUILD process involves linking in library modules. Your new memory disk code will be placed in a library file called USER.LIB.

The BUILD program will look for this file when it links, and include your drivers into the ROM-DOS kernel.

When you run the BUILD program, you are prompted to include a custom memory disk. Answer yes for this prompt to add the new built-in driver. You must have a USER.LIB file present in the same directory as the BUILD program when including a custom memory disk, or errors will be generated during the compilation processes.

Memory Disk Base

The memory disk base module is the non-changeable part of the memory disk. The base module may be configured using assembler options. Available options for the base are placed in the top lines of code in MEMDISK.ASM. The MAKEFILE already defines the necessary options for both stand-alone and built-in versions of the drivers. Additional options can be added to the MAKEFILE or entered on the command line. If you customize the options, the options set in the MEMDISK module must be in agreement with the options set in the client code module. Most of these configuration options follow the standard conventions of undefined = false or off, and defined = true or on. The exception is STACKSIZE, in bytes, which defaults to 1024.

About Client Code Functions

The client code module of the memory disk must be ported to the different environments in which it is to be used. The functions can be coded in any language but they must conform to the C-language calling sequences and conventions. The client code module(s) must supply some of the functions listed below, depending on their purpose.

A ROM disk requires only the **meminit** and **memread** functions while a RAM disk requires these plus **memwrite**. All other functions are optional.

TSR disks must support **memuninit**, removable disks must support **memchanged**, while disks that support IOCTL must support the **memioctl** function.

meminit

The **meminit** function is called once during disk initialization.

```
int meminit(struct BPB bpb[], char far *cmdline, unsigned *endseg,
int drv);
```

The BIOS parameter block, pointed to by the argument *bpb*, should be filled in during **meminit**. The *cmdline* argument is used for parsing the device command line. This pointer points to the start of the memory disk filename in the CONFIG.SYS device line. The *cmdline* argument has no meaning for built-in devices, but can be useful for devices installed in CONFIG.SYS. The return value of **meminit** is the number of drives.

The *endseg* argument, when first called, points to a value that is the next available segment, the segment used for a RAM disk if DOS RAM is to be used for the disk. If memory at the *endseg* is being used, *endseg* must be updated to account for the amount of memory used. If DOS RAM is not to be used, or this device is to be built-in, then this argument is ignored.

A built-in RAM disk must allocate memory using the DOS memory allocate function (Int 21h, AH=48), in the **meminit** function. This is how a built-in RAM disk would get DOS RAM for disk storage.

Note: This is used only for built-in devices, not those that are CONFIG.SYS loadable. Installable devices should update *endseg*.

memread and memwrite

```
bool memread(long offset, unsigned len, char far * buffer);
bool memwrite(long offset, unsigned len, char far * buffer);
```

The **memread** and **memwrite** functions use a 32-bit value, named *offset*, to specify where on the disk a read or write operation occurs. This value is usually, but not always, a multiple of the sector size. The *len* argument is a 16-bit unsigned number that defines the size of the read or write in bytes. The *buffer* argument is a 32-bit far pointer that defines where the disk data is read from or written to. The return value from **memread** and **memwrite** is a non-zero for success and zero for failure. A failed return value causes a critical error.

memchanged

The **memchanged** function notifies DOS that a removable disk has been removed.

```
int memchanged(struct BPB *bpb)
```

memchanged is a C-callable device driver media-check.

The value returned from the **memchanged** function indicates if the disk has changed or not. A return value of -1 indicates that the disk has changed while a return value of 1 indicates that the disk has not changed. If the memory disk is not a removable disk, **memchanged** should always return 1. If the memory disk is a PCMCIA memory card that can be removed, this function is critical.

A return value of 0 from **memchanged** indicates that the disk may have changed. A 0 return value is acceptable but not recommended as most modern BIOSs today support a change-line-status-function that returns a definite status. (The 0 return value was used on the original PC, which did not support such a BIOS call.)

memuninit and memioctl

memuninit and **memioctl** have the following syntax:

```
void memuninit(void)
bool memioctl(unsigned category, char far * buffer);
```

Public Fields

There are also two data fields are defined in MEMDISK.ASM that are intended to be used or set by the client code:

```
unsigned char memerr;
unsigned char memunit;
```

The client code should set *memerr* when an error occurs during any of the **memread**, **memwrite** or other client functions. *memerr* is ignored if no error occurs so there is no need to reset or clear

it. The *memunit* field is set by MEMDISK.ASM before passing control to the client code. The unit number will be in the range of 0 to *n* where *n* is the number of drives returned by **meminit** upon driver initialization.

Terminate-and-Stay-Resident (TSR) Drivers

You can configure a custom disk driver so that it can load from the DOS prompt or from a batch file, as well as from CONFIG.SYS. The TSRDEV switch in MEMDISK.ASM should be set to true for this option. A custom disk driver installed at the DOS prompt can also be unloaded from memory, thereby freeing the memory and drive letters occupied by that driver. The following functions and data are defined only in the stand-alone TSR-enabled custom disk driver:

```
void tsr_setidstr(char * tsr_id);
int tsr_uninstall(void);
int memdev_already_loaded(void);
find_free_drives;
```

The function **tsr_setidstr**(char * tsr_id) allows your custom memory disk to have a unique identifying sequence of bytes. You can also retain the default identification sequences. Each custom memory disk driver should have a unique identifier.

Use **tsr_uninstall** to remove the latest driver from memory. **tsr_uninstall** fails if the device is not found or was loaded from CONFIG.SYS instead of from the command line.

Use **tsr_already_loaded** to determine if a previous copy of the custom disk driver has been installed. Multiple copies are allowed as a convenience in the event your driver implementation needs to check for another disk already resident.

The **find_free_drives** function checks both CDSs and DOS drive #s to determine the next sequential DOS drive number to use.. That is, if LASTDRIVE is set to E: and drives A:, B:, and C: are used, there are two drive letters available (D: and E:).

Memory Disk Math Routines

For built-in device drivers, ROM-DOS includes a set of supported math functions. The MEMDISK.H module includes those functions not already built-in to ROM-DOS. The built-in ROM-DOS functions, plus the math routines in MEMDISK.H, replace the math library normally linked in at compilation time. Memory disk client code that will be built-in should not use long math but equivalent functions. The supported math functions and their definition prototypes include:

```
typedef unsigned long ulong;
typedef unsigned char uchar;
typedef int bool;
ulong pascal _lshru(ulong l, unsigned u);
ulong pascal _lshlu(ulong l, unsigned u);
ulong pascal _lmulu(ulong l, unsigned u);
ulong pascal _ldivu(ulong ul, unsigned u);
ulong _ldiv(ulong lVal, ulong lDivideBy);
ulong _lmod(ulong lVal, ulong lModBy);
void pascal memmove(void *to,void *from, unsigned len);
void pascal fmemcpy(void far *dst,void far *src, unsigned len);
void pascal memset(void *dst, uchar val, unsigned len);
```

```
void pascal fmemset(void far *dst, uchar val, unsigned len);int
pascal strlen(char *s);
int pascal fstrlen(char far * s);
        // Note: Unlike the standard C memcmp, this memcmp returns
        //       TRUE if strings are identical, FALSE if different.
        //       Same for fmemcmp.
int pascal memcmp(char *str1, char *str2, unsigned len);
int pascal fmemcmp(char far *str1, char far *str2, unsigned len);
```

Making Special Configuration Changes

While the BUILD program allows complete and easy configuration of ROM-DOS for most installations, there may be designs that require you to make changes to SYSGEN.ASM prior to running the BUILD program. This chapter covers the areas of system configuration that may need special attention to accommodate your design.

In addition, this section describes how boot-time configuration can be controlled through the standard CONFIG.SYS file or by reconfiguring the BIOS to change the way ROM-DOS operates.

Configuring ROM-DOS Through SYSGEN.ASM

The file SYSGEN.ASM allows you to configure the operation of ROM-DOS at compile time. Most of the options in SYSGEN are configured by answering prompts when running the BUILD program as described in Chapter 4. The following configuration options are described in subsequent sections; you can modify their behavior without having to read the source code.

- Assembly defines
- List of built-in devices
- Power save option
- CONFIG.SYS defaults
- Initial environment
- ROM disk search address

Assembly Defines

The assembly defines configure ROM-DOS in SYSGEN.ASM. The BUILD program normally sets these as it assembles the SYSGEN.ASM file. The defines are described in the following table.

Option	Description
BCHECK=1	Display boot diagnostics. BUILD will set or clear this option as appropriate.
BEXT=1	Boot from BIOS extension; otherwise it will boot from disk. BUILD always sets this option.
BOOTDEV=id	ROM-DOS can boot from any device (floppy, hard or ROM disk). The id code specifies which one: 00=floppy,

Option	Description
	80H=hard, 10H= ROM disk.
BOOTDRV=n	ROM-DOS can boot from a specific driver letter. Choose the boot drive letter where 0=A, 1=B, 2=C, and so on.
BOOTMENU=1	Display the following menu upon boot time. The menu lets the user choose from where to load DOS, and/or where to read CONFIG.SYS.

```
ROM-DOS boot options:
1. Load DOS off floppy
2. Load DOS off hard disk
3. Make floppy default drive

4. Make hard disk default drive
5. Make ROM disk default drive
6. Continue as if Alt key not pressed
```

DATASEG=seg	Hard-code the ROM-DOS DATA segment (no fix-ups). Otherwise LOC will set the DATA segment address. Set seg equal to the destination segment in RAM. This value can be either hex or decimal.
FLOPCFG=1	Regardless of boot drive, ROM-DOS looks for CONFIG.SYS on the floppy disk before going to the ROM disk.
FLOPCHK=1	A floppy disk DOS can supersede ROM-DOS in ROM.
GENERIC=1	Include generic custom memory disk driver.
HARDCHK=1	Bootable hard disk partition can supersede ROM-DOS in ROM.
NOFLOP=1	No floppy or hard disk is needed (ROM disk only). This can save approximately 3KB of ROM.
NOROM=1	No ROM disk is desired This can save approximately 1KB of ROM/disk.
RAMBOOT=seg	Copy ROM-DOS from ROM to RAM upon boot for faster execution. Set seg equal to the destination segment in RAM. Default for BUILD.EXE is 70h. This value can be either hexadecimal or decimal.
RDISK=seg	Choose the segment where the built-in ROM disk driver starts searching for the ROM disk.
USERTC=1	Use the real-time clock exclusively instead of the BIOS ticks for the time.

The following example shows the use of some assembly defines/options.

```
TASM /Mx /DEXT=1 /DBOOTMENU=1 SYSGEN;
```

The BUILD program automatically generates the above assembly command-line given the appropriate options. There is usually no need to assemble SYSGEN.ASM manually.

The example configures ROM-DOS to boot from a BIOS extension. It also causes ROM-DOS to display a boot menu if the user holds down the Alt-key during boot up.

Listing Built-in Devices

Built-in device drivers can be added to ROM-DOS by placing the name of the device driver header in a list of block device drivers terminated by a NULL (0) character. A new device may be added by placing its PUBLIC label name in the list.

Note: The block device drivers list must have at least one disk device entry so that the file CONFIG.SYS and the initial program can be read from disk at boot time.

The only disk device required in this list, for an embedded system, is the ROM disk driver. This device driver is supplied in source form on the distribution disk. The PUBLIC name for this device is *–romx* and it is found in the file DEVROM.ASM.

The code and data size of ROM-DOS can be decreased by commenting-out external references to unused devices in SYSGEN.ASM. If no references are made to a device in SYSGEN.ASM, the device will not be linked in from the library. The size of ROM-DOS decreases by the amount of code/data space occupied by that device driver.

The following example shows the list of built-in devices listed in SYSGEN.ASM:

```
public      _built_in
built_in    LABEL WORD
; built-in character devices
  dw OFFSET DGROUP:_nulx    ; MUST be 1st
  dw OFFSET _DEVDATA:_conx  ; MUST be 2nd
  dw OFFSET _DEVDATA:_clkx  ; MUST be 3rd
  dw OFFSET _DEVDATA:_comx  ; not required
  dw OFFSET _DEVDATA:_prnx  ; not required
  ; warning: while the COM and PRN(LPT) drivers are not required, the
  ; absence of them can cause somewhat odd behavior in some programs.
  ; built-in disk devices (at least 1 disk required)
IFNDEF NOFLOP
  dw  OFFSET _DEVDATA:_fdhdx ;  optional
ENDIF
IFNDEF NOROM
  dw  OFFSET _DEVDATA:_romx  ;  optional
ENDIF
IFDEF GENERIC
  dw  OFFSET _DEVDATA:_memx  ; optional
ENDIF

  dw  0  ; NULL terminator for list
```

Note: SYSGEN.ASM has two complete lists of the device drivers. One list is defined with the "dw OFFSET" syntax as shown in the above listing. The other list is defined with the

"extrn" syntax and appears in SYSGEN.ASM immediately before the above list. You must add or remove drivers of the same name in both the "dw OFFSET" and "extrn" lists.

Power Save Option

ROM-DOS, when not actively performing an application function, spends much of its time waiting for user input. During that time, ROM-DOS checks the BIOS for another character, performs an Int 28h, and then goes back to checking the BIOS for a character. Even when there is no user input, the computer is using electrical power. It is possible to avoid this waste of power on computers that support a static state or a slower processor speed.

To use the power save option, the BIOS Int 16h, function 00h is modified to switch into low power mode until a key is pressed. A *powersave* flag causes ROM-DOS to either poll the BIOS or call it and wait. If *powersave* is 0, then ROM-DOS polls the BIOS (Int 16h Function 1). If *powersave* is set to 1, ROM-DOS calls the BIOS and waits (Int 16h Function 0).

CONFIG.SYS Defaults

This section of SYSGEN.ASM allows you to define the default number of FILES, BUFFERS, the status of BREAK, and most options you normally set in CONFIG.SYS. However, Datalight does not recommend modifying this section since it is easier to modify CONFIG.SYS. If your system does not use a CONFIG.SYS file, it may be necessary to set these values in the SYSGEN.ASM file.

One such CONFIG.SYS default is the command interpreter COMMAND.COM, the first program executed by ROM-DOS. The initial program for an embedded system could be your application program. SYSGEN.ASM allows you to set the program name and the initial command line argument string. Both strings must be null terminated.

```
c_public         init_break,init_files,init_buffers
c_public         init_fcbs,init_lastdrive,init_shell
_init_break      db 0                    ; BREAK=OFF
_init_files      dw 8                    ; FILES=8 (0=calc)
_init_buffers    dw 0                    ; BUFFERS= (0=calc)
_init_fcbs       dw 2                    ; FCBS=2
_init_lastdrive  db 'E'                  ; LASTDRIVE=E
_init_shell      db "COMMAND.COM /P",0   ; SHELL=COMMAND.COM /P
```

Also in this section are settings for the position of the Confirmation message when using the MENU commands with Config.sys and the keystoke definitions if a user wants to use alternate choices from the traditional F5 and F8 commands. In addition, there is a setting for changing the position of the timeout counter displayed when using a MENUDEFAULT command.

The DOS Version

The DOS version may be set to anything desired, but be aware that all the internal structures work like DOS 6.x regardless of the version reported.

```
public dos_ver
```

```
IF USING_FAT32
_dos_ver  label  word
   db     07H    ; AL = major version (DOS 7.1)
   db     0AH    ; AH = minor version (7.10)
else
_dos_ver  label  word
   db     06H    ; AL = major version (DOS 6)
   db     16H    ; AH = minor version (6.22)
ENDIF
```

The Initial Environment

The initial environment string and maximum size (in paragraphs) are set in SYSGEN.ASM. There may be multiple environment variables, each separated by a zero-byte. The end of the environment is determined by a second zero-byte. The variable *env–para*, is where the number of environment paragraphs is specified. This value must be larger than the space required to hold the initial environment string.

```
c_public        env_para,env_string
_env_para       dw 10H
_env_string     db "PATH=",0
                db "PROMPT=$p$g",0
                db 0
```

The ROM Disk Start Address

The start address of the ROM disk is specified with a 16-bit segment value named *romdisk*. This value represents the first segment at which the ROM disk driver looks for a valid ROM disk. The driver searches ROM for the ROM disk until it finds a valid disk or reaches the end of memory (segment 0xFFFF). The following example causes the ROM disk driver to begin its search at segment C000 and search until it reaches 0xFFFF.

```
IFDEF RDISK
romdisk   dw   RDISK ; segment of ROM disk
ELSE
romdisk   dw   0C000H ; segment of ROM disk
ENDIF
```

The BUILD program prompts for a change in the default search segment. Datalight recommends that you do not make a change to the segment in SYSGEN.ASM, but let BUILD handle it.

Configuring Through CONFIG.SYS

ROM-DOS supports the standard system configuration file, CONFIG.SYS. This file contains commands that reconfigure the system during boot-up. The CONFIG.SYS commands supported by ROM-DOS include:

```
BREAK=            MENUDEFAULT=
BUFFERS=          MENUITEM=
COUNTRY=          NEWFILE=
DEVICE=           NUMLOCK=
DEVICEHIGH=       REM=
DOS=              SET=
FCBS=             SHELL=
```

```
FILES=              STACKS=
INCLUDE=            SUBMENU=
INSTALL=            SWITCHES=
LASTDRIVE=          ;
MENUCOLOR=          ?
```

The NEWFILE command is unique to ROM-DOS and allows CONFIG.SYS to transfer control to another CONFIG.SYS file, possibly on some other drive or in a subdirectory. Use the NEWFILE command as follows.

```
NEWFILE=filename.ext[,driver.sys [parameters]]
```

You can use this command to pass control to a new drive installed via CONFIG.SYS as shown in the following example.

```
NEWFILE= NEWCFG.SYS,NEWDISK.SYS E:
```

When ROM-DOS is configured with the BUILD program, you can select from the following four levels of CONFIG.SYS processing. These options are available only if your kernel does not support LFNs

DOS Level	Commands Included
3.31	BREAK, BUFFERS, COUNTRY, DEVICE, FCBS, FILES, LASTDRIVE, NEWFILE, REM and SHELL.
5.0	All commands available with DOS 3.31, plus DOS, INSTALL, and STACKS.
6.0	All commands from both the DOS 3.31 and 5.0 levels with the addition of DEVICEHIGH, INCLUDE, MENUCOLOR, MENUDEFAULT, MENUITEM, NUMLOCK, SET, SUBMENU, and SWITCHES, the semicolon (;), and the question mark (?).
7.1	This is the same as DOS 6.0

ROM-DOS Long Filename Support

ROM-DOS now optionally contains Windows 98-style long filename support in the kernel. As indicated in “Configuring Through CONFIG.SYS” on page 176. You must have enabled LFNs when building ROM-DOS in order to utilize this support.

ROM-DOS operating system provides support for the following long filename functions. The longname disk layout is fully compatible with Windows 98 long filenames.

Int 21h

5704 - Get Last Access Date/Time
5705 - Set Last Access Date/Time
5706 - Get Creation Date/Time
5707 - Set Creation Date/Time
7139 - Create Longname Directory
713A - Delete Longname Directory

713B - Set Current Working Longname Directory
7141 - Delete Longname File
7143 - Get/Set Longname Attributes/Dates
7147 - Get Current Working Longname Directory
714E - Find First Longname
714F - Find Next Longname
7156 - Rename Longname File or Directory
7160 - Get Longname Path
716C - Open/Create Longname File
71A0 - Get Longname Volume Info
71A1 - Find Close
71A6 - Get File Info By Handle

Please note that the undocumented function 71A8, Get Longname Alias, is no longer supported by MS-DOS; consequently there is no support for it in ROM-DOS either.

Configuring Through the BIOS

Another possible method of configuring ROM-DOS is to change the BIOS to handle new hardware while leaving the normal device drivers intact. For example, you could modify Int 13h of the BIOS so that that the floppy and hard disk drivers operate on some different disk hardware, such a PCMCIA memory cards.

The advantage of modifying the BIOS, especially if you have your own BIOS that you are familiar with, is the time saved in writing and debugging a new device driver. In many cases, the standard drivers will work normally through a modified BIOS.

Creating a Custom Sign-on Message

ROM-DOS allows for flexible sign-on messages. The standard "Starting ROM-DOS..." message can be customized for use with special evaluation kits, or to allow alternative sign-on screens. To make your own sign-on, follow these steps:

1. Modify the "starting_msg" string in DOSIGNON.C, found in the DEVSRC directory of the installed SDK.
2. Compile DOSIGNON.C to DOSIGNON.OBJ (linking is not necessary), for example
   ```
   bcc -c dosignon.c
   Or, if you are using code defines
   bcc -c -DMYMSG=1 -DEVALKIT=1 dosignon.c
   ```
3. Place DOSIGNON.OBJ in USER.LIB using Borland's TLIB command. Refer to '**Including Device Drivers**' for an example of using the TLIB command. Please note, the TLIB Librarian utility provided with the Datalight SDTK requires 32-bit DPMI support. Running TLIB from a Win95/Win98 DOS box or using a memory manager that provides DPMI support will be required.
4. Run BUILD to create a version of ROM-DOS with the new sign-on message. The new sign-on replaces the "Starting ROM-DOS..." message.

The Command Interpreter

The command interpreter loaded by ROM-DOS may be specified with the SHELL command in the SYSGEN.ASM or CONFIG.SYS file. The command interpreter is normally the COMMAND.COM program.

```
SHELL=COMMAND.COM /p /e:512
```

For many embedded systems, a command interpreter may not be required. Any program can start at boot time and have full use of ROM-DOS, as is usually the case with single-application systems.

By specifying a command interpreter other than COMMAND.COM, the ROM or disk space (about 45KB) and RAM space (about 3KB) required by COMMAND.COM can be saved. However, without COMMAND.COM loaded, the DOS prompt and the processing of batch files (including AUTOEXEC.BAT) are not available. Optionally, Datalight provides a mini-command interpreter that supports a limited command line and batch processing.

The command interpreter loaded at boot time must adhere to the following rules.

- It must never terminate. If it does terminate, ROM-DOS prints a message indicating that the command interpreter has quit and then halts the system.
- It must handle Ctrl+C if it can occur. If Ctrl+C is encountered and it is not handled, then the shell program is terminated and the system halts.
- It must handle Int 24h (critical error handler) if it can occur. If a critical error is encountered and it is not handled, the shell program is terminated and the system halts.

Debugging and Troubleshooting

ROM-DOS provides standard MS-DOS functionality in the ROM environment. This allows most of the actual program development to be performed on a desktop PC running DOS. The remainder of the development can be done on the target hardware under ROM-DOS. The routines that most likely require debugging under ROM-DOS are those device drivers and other program segments that access non-standard, non-PC hardware. This section provides information and solutions about problems that may occur during the startup process for ROM-DOS on your system. Such problems may also be attributable to your BIOS.

Print Statements

The simplest method for debugging your program is running your program with embedded print statements at meaningful points. This method of debugging requires a console available on your target system. The console may be a serial port or a display/keyboard combination.

The program can be uploaded to a target system RAM disk using the COMM or TRANSFER programs. The TRANSFER program takes a file from the host PC, across the console, and places the file on a RAM disk or other disk device on the target system. Refer to the ROM-DOS User's Guide for more information on the TRANSFER and COMM programs.

Remote Debugging

The Borland Turbo Debugger provided in the Datalight STDK can be used in the remote mode if there are COM1 or COM2 serial ports available on the system. TD-REMOTE provides an ideal interface and flexible debugging. For more information, refer to the Borland help files.

Local Debugging

If your target hardware has a PC-compatible display and keyboard, you can use your normal debugger on the target hardware under ROM-DOS. Some debuggers check for and require a particular DOS version number. You can use the VER command (refer to the ROM-DOS User's Guide) to change the version number reported by ROM-DOS to that required by your debugger.

Troubleshooting with Boot Diagnostics

ROM-DOS has the ability to display special characters at each stage of the boot process. These characters provide a method for determining where in the boot process an error occurs. These characters are referred to as boot diagnostics and are included in the ROM-DOS kernel if you answer Y to the following prompt displayed by the BUILD program.

```
Do you want boot Diagnostics (Y/N): Y
```

To perform a manual link of ROM-DOS, assemble the file SYSGEN.ASM with the option /DBCHECK=1 enabled. See ', Making Special Configuration Changes' for details on manually linking ROM-DOS.

The boot diagnostics are displayed (via BIOS Int 10h, function 0Eh) to indicate completion of each step of the boot process. The boot process steps are listed below.

Boot Diagnostic	Description
B	BIOS extension has gained control. This is only displayed when ROM-DOS is placed in ROM. When booted from a disk, this boot diagnostic is not shown.
0	Interrupts are enabled, ROM-DOS has control, and the first instructions have been executed.
1	Startup code (decompress) has completed. The startup code copies the DOS data into RAM. The DOS code is also copied for a disk boot of ROM-DOS or a ROM boot with the copy to RAM feature enabled. To make room for data decompression, the startup code relocates the ROM-DOS code to the top of memory. The data is decompressed to its full size in lower memory. The stack is also set up and uninitialized data is zeroed. Boot failures at this point are typically due to insufficient RAM to accommodate the code and full data size, or an incomplete ROM-DOS image in ROM.
2	Minimum DOS structures allocated. The memory pool is set up and the default structures are at the top of RAM. The DOS interrupts are also set up.
3	Interrupts have been initialized. Boot failures at this point may be due to another process using an interrupt that ROM-DOS has set up for its own use. An example of this is a watchdog timer that traps Int 21h.
4	Built-in devices have been initialized. BIOS interrupt calls are made during the initialization (Int 13h for disk drive support, Int 10h for video). Failures may be

Boot Diagnostic	Description
	due to incomplete BIOS interrupt support or a failure to find a disk of any type in the system.
5	Root PSP is now in existence.
6	Default drive has been determined.
7	The first pass of CONFIG.SYS processing is complete. All CONFIG.SYS statements except the INSTALL= are processed (device drivers are loaded). Standard handles such as PRN, AUX, and CON are opened.
8	All internal structures allocated. TSR programs listed in CONFIG.SYS INSTALL= statements are loaded. Failure at this point may indicate a faulty TSR program.
9	ROM-DOS has been loaded high (if DOS=HIGH). The DOS buffers have been created and copied to the HMA area if sufficient space.
DOS prompt or application start	The standard handles have been re-opened and the final programhas been called via Int 21h. This program is typically COMMAND.COM or an application program. Failure to reach the DOS prompt after boot diagnostic 9 is usually caused by not finding the program (either not on the disk or a corrupted file), a command interpreter from a different operating system, or a faulty application program. Failure may also be due to insufficient RAM to run the command interpreter or application program.

Some Common Problems

This section lists some of the more commonly encountered problems. Refer to the Support section on our Datalight website for additional information, white papers, and links to our technical support.

Problem: During the boot process the following error message appears:

```
No Disk Devices System Halted
```

Solution: The ROM-DOS kernel did not find a disk containing of any sort in the system. Check the SYSGEN.ASM file for the block device list. Be sure that there is at least one disk in the list.

If the system has only a ROM disk, then the ROM disk driver was unable to find it in memory. Check the SYSGEN.ASM file for the segment address from which the ROM disk started its search for. Look at variable *–romdisk* in file SYSGEN.ASM.

Problem: During the boot process the following error message appears:

```
ROM-DOS Not Found, System Halted
```

Solution: The BIOS did not find the ROM-DOS BIOS Extension.

The ROM-DOS BIOS extension must be placed in the addressable memory. If this is not the case, the BIOS does not search low enough in memory for the ROM-DOS BIOS extension.

Problem: During the boot process the following error message appears:

```
BAD or Missing ProgramName
```

Solution: The *ProgramName* was not found on the default disk or it was not loadable. Check the *init–prog* variable in the file SYSGEN.ASM and check that the file is on the disk.

Once all options and devices have been set and ROM-DOS has been linked and located, it is time to program ROM-DOS into ROM. There may be up to three files connected with ROM-DOS that are programmed into ROM:

ROM-DOS.HEX - the ROM-DOS kernel

ROMDISK.HEX - the ROM disk image

BIOS.HEX - the BIOS (optional)

The three files listed above have the .HEX extension for Intel hex files, but could also be .IMG binary image files. The first file, ROM-DOS.HEX, is approximately 56 to 76KB unless significant device support has been added or removed. The second file, ROMDISK.HEX, allows for booting on a completely diskless system and can range from 1KB to just under 1MB in size. A practical limit is usually 512KB for conventional memory installation.

The optional BIOS is designed to fit in a minimum of ROM along with ROM-DOS. Although the BIOS.HEX is optional, ROM-DOS does require a BIOS to run. Since some hardware has a pre-installed BIOS, you may not have to program an EPROM with the BIOS. For a pre-installed BIOS, it is helpful to know its size and memory location so that you can properly locate ROM-DOS.

A typical BIOS starts searching for a BIOS extension signature starting at address C000:0H and then continues the search on every 2KB boundary up to address F000:0H or F800:0H. This allows ROM-DOS to be placed in ROM anywhere between C000:0H and F000:0H. Under unusual circumstances, ROM-DOS can be placed outside this address range and be initiated via a special BIOS extension program.

As ROM-DOS boots, it searches for Datalight ROM disk signatures in memory (assuming you have included a ROM disk driver in your configuration). By default, ROM-DOS searches for ROM disks within the same search range as the standard BIOS. ROM-DOS searches only on 16-byte paragraph boundaries. A 16-byte boundary is represented by an address *nnnn*:0 where *nnnn* indicates a value in the default range of C000H to F000H. However, ROM-DOS is flexible enough to allow the search range to be expanded.

You can specify a new starting point for the search when you run the BUILD program to configure ROM-DOS. The ROM disk can be located either above or below ROM-DOS in conventional memory or, with the addition of specially modified drivers, in extended or paged memory.

Creating ROMable Applications

In many embedded computer systems, mechanical disk drives are emulated by a combination of data structures and code that are contained in ROM. DOS ROM disks exist within the executable

first megabyte of address space of Intel processors operating in Real Mode. Because programs routinely need to update data as part of their normal execution, an ordinary .COM or .EXE file must first be copied to RAM by DOS before the program can executed. When execution space is at a premium, these duplicated ROM and RAM images are wasteful of system resources. If a strategy could be developed to execute at least part of the program directly from ROM, without copying that portion of the program image to RAM, then embedded system manufacturers could realize significant space savings. The Datalight ROMable EXecutable (RXE) conversion splits the program parts and changes the program's code to Execute in Place (XIP), thus saving RAM.

For more information, please refer to the document file **RXE Theory of Operation.PDF** in the RXE directory of your installed SDK.

RXE Convert Operation

The RXE convert (RXE_CVT.EXE) utility will modify a program in such a way that the program's code will execute out of ROM but the program's data will be allocated at execution time by DOS. Therefore, the code segment for the program will be known for an RXE program, but the data segment will not be. Each of the fixups must be evaluated and handled specially by RXE_CVT.

Syntax

RXE_CVT *[/C] [/Ixx] [/Q] [/R] [/S] [/W] [/L] InFile DataSegName MemorySeg [OutFile]*

Remarks

The RXE verify tool (RXEVERFY) should be used to verify all conversions.

Options

/C Tells the conversion to continue automatically if errors found.

/Ixx Allows the programmer to set the RXE interrupt number (in hexadecimal).

/Q Quiet mode.

/R Names the OutFile with an extension of .RXE.

/S Used to get the RXE file size.

/W Displays all warning messages.

/L Should be used if your .MAP file has mixed or lower case segment names.

RXE Optimize Operation

RXE optimize (RXEOPTIM.EXE) is a utility designed to reduce the size of an RXE program by removing some of the unused space in the file. When RXE_CVT creates an RXE program, the size of the fixup list is retained even though most of the fixup entries have been removed. Furthermore, most compilers allocate more space than is necessary for the original fixup list.

To optimize the RXE program, RXE optimize must first determine how much unused space is in the EXE file. It will then remove the unused space and adjust all the remaining fixups, which have already been performed by RXE_CVT, accordingly. Finally the new entry points are computed and a new checksum is performed.

Syntax

RXEOPTIM *exename rxename*

Remarks

The RXE verify tool (RXEVERFY) should be used to verify all optimizations.

RXE Verify Operation

RXE verify (RXEVERFY.EXE) is designed to re-evaluate all the work performed by RXE convert and RXE optimize. This utility looks at each fixup and makes sure that the fixup case was handled appropriately. It will also ensure that no unnecessary modifications were made to the rest of the program.

Syntax

RXEVERFY *[/O] exename rxename*

Remarks

RXE verify can be used to verify the results of both RXE optimize and RXE convert.

Options

/O Forces RXE verify to detect the RXE input file as an RXE optimized file.

Power Management

Overview

As more and more computers become mobile, batteries become the desired power source. Power management is a critical issue for battery-powered systems. Power management requires cooperation between applications, DOS, the BIOS and the hardware. Intel and Microsoft have defined a DOS/BIOS level of cooperation that an application can query. This specification, named Advanced Power Management (APM), involves a TSR program (POWER.EXE, hereafter referred to as POWER) provided by Datalight, which communicates to the BIOS and applications.

POWER provides a real mode APM 1.1 connection to the APM BIOS and complies with all requirements of the appropriate specification – see the APM 1.2 Specification 'Appendix D – APM Driver Considerations.' The specific functionality of both the application and BIOS interfaces are explained in the paragraphs that follow. Copies of the APM 1.2 BIOS interface specification may be obtained by searching for "APM 1.2" at the following locations on the World Wide Web:

```
http://www.microsoft.com
http://www.intel.com
```

Operation of POWER.EXE and the Application Interface

As an APM driver, POWER allows application programs to notify it when they are idle and to reject requests by other applications, by POWER itself or by an APM BIOS to transition the system into a lower power state. POWER also provides a mechanism by which application programs can be notified of changes in system power availability.

POWER interfaces with DOS application programs using a two-phase software interface. This interface employs interrupt 2Fh and allows a DOS program to:

- Notify the system that it is idle and initiate a request to reduce the system power level.
- Receive notification of changes in the system power availability.
- Reject requests by other applications, by POWER or by the APM BIOS to reduce the system power level.

While POWER does monitor a number of devices for activity that precludes reducing system power availability, it does not provide power management for individual devices. POWER always reduces the available power state for the CPU and for all devices which may be controlled by the APM BIOS. This feature does not preclude an APM BIOS from reducing the power level of individual devices (either automatically or cooperatively), but POWER does not initiate such power reductions. If an APM BIOS generates a request to reduce a specific device's power level, POWER broadcasts the event to any cooperating applications. If no applications reject the device specific power down request, POWER calls the BIOS to change the device's power state.

Notifying the System that the Application is Idle

Applications notify POWER that they are idle by loading the AX register with the value 1680h and then generating Int 2Fh. No status is returned by the POWER interrupt handler and no registers are changed by POWER.

Note: Int 2Fh is an often-called interrupt that may be used by many programs. POWER cannot guarantee that another program in the Int 2Fh chain will not change any register values.

Receiving Notification of System Power Changes

To receive notification of APM events, an application must hook Int 2Fh and process calls in which the AX register contains the value 530Bh. On receipt of such an interrupt, the BX register contains an APM event code. POWER acts only on those events listed below.

Event	Event Type	Event Code
SYSTEM_STANDBY	Request	0001h
SYSTEM_SUSPEND	Request	0002h
END_NORMAL_SUSPEND	Notification	0003h
END_CRITICAL_SUSPEND	Notification	0004h
BATTERY_LOW	Notification	0005h
POWER_STATUS_CHANGE	Notification	0006h
UPDATE_TIME	Notification	0007h
CRITICAL_SUSPEND	Notification	0008h
USER_STANDBY	Request	0009h
USER_SUSPEND	Request	000Ah
END_NORMAL_STANDBY	Notification	000Bh

POWER responds to each of the above APM events with the actions defined in the APM 1.2 specification. POWER transmits any event code returned by the APM BIOS to any APM applications that have hooked the Int 2F chain. Applications, and, device drivers may augment the functionality of POWER by processing these additional events and either controlling the availability of specific devices directly or initiating application-idle signals as appropriate.

Whenever an application receives a notification event or accepts a power change request it should pass the event onto the next handler in the Int 2Fh chain without altering any of the original register values.

Rejecting Requests to Change the Current Power State

Applications may reject any of the request event types defined in the table on page 186. To reject an APM request event, the application sets the BH register to 80h and immediately returns from the Int 2Fh call without altering any other register values.

It is possible for other APM-aware applications to be running that have early positions in the Int 2Fh interrupt chain. It is also possible for such applications to have already saved their operating state in response to a power-down request before a subsequent APM application rejects the request. POWER generates a resume notification event following any rejected power-down request to allow any applications, which may have acted on the request, to return to a fully-operational state.

The BIOS Interface to POWER

When POWER detects an APM BIOS that supports version 1.1 or later of the APM BIOS Specification, it initiates a version 1.1 connection to the APM BIOS in real mode. As a client of the APM BIOS, POWER acts only on those event codes defined in the table on page 186. POWER complies with the APM 1.2 Specification and uses the following APM BIOS functions via the software Int 15h interface.

APM BIOS Function	Function Number
APM_INSTALLATION_CHECK	5300h
APM_REAL_MODE_CONNECT	5301h
APM_INTERFACE_DISCONNECT	5304h
APM_SET_POWER_STATE	5307h
APM_RESTORE_DEFAULTS	5309h
APM_GET_PWR_STATUS	530Ah
APM_GET_PWR_MGMT_EVENT	530Bh
APM_GET_DRIVER_VERSION	530Eh

All events initiated by POWER are system level power requests. No specific devices are supported, and consequently, neither is the APM_ENABLED state. POWER monitors the BIOS software interface to the disk, serial ports, keyboard, printer ports and display, as well as the hardware keyboard interrupt, for activity. POWER uses the periodic software Int 1Ch to measure

the time since the last user activity and to poll the APM BIOS for pending events. If no events occur within the least-time-out value specified on the POWER.EXE command line, POWER generates a power-down request.

POWER also generates a power-down request in response to an application-idle request. In either case, if no applications reject the power-down request, POWER calls the APM BIOS to set the appropriate system power state.

Note: POWER only calls the BIOS to reduce the current power availability, never to increase it. The BIOS is responsible for increasing the power availability and for notifying POWER (by posting an APM event) that the power availability has changed.

If the system power state is initially in the ready or full-power state, POWER attempts to set the system power state to standby. If the system state is already in the standby state and a time-out occurs due to no user activity (or if a subsequent application idle event is received), POWER attempts to further reduce the current power state to the suspend state.

The APM BIOS must notify POWER of any increase in power availability by posting an APM event. Whenever POWER processes such events, it automatically sets the timer tick count to the time kept by the CMOS real-time clock (if one is available). This strategy of directly setting the system power state is the only method POWER employs to control power consumption. No CPU_IDLE or CPU_BUSY calls are generated by POWER.

POWER transmits any APM BIOS event code (supported or not) to any APM applications that have chained into Int 2Fh. DOS applications and device drivers may extend POWER's functionality by processing these additional event codes and either controlling the availability of specific devices directly or generating application-idle signals.

Installation and Usage

POWER can be loaded either at system startup or when the system is running. Once loaded, POWER remains in memory and active until the system is turned off. POWER can be loaded at system startup by placing an INSTALL command in the CONFIG.SYS file as shown in the following example.

```
INSTALL=C:\BIN\POWER.EXE
```

POWER can also be installed by means of a statement in the AUTOEXEC.BAT file.

Operation of POWER can be configured using the following command line options. The # sign defines the number of seconds that a device may be inactive before it is powered down.

/C#	Set the inactive time for COMM ports.
/D#	Set the inactive time for disks.
/H	Display the basic help screen.
/K#	Set the inactive time for the keyboard.
/P#	Set the inactive time for printers.
/S#	Set the inactive time for the display.

/ADV:MIN	Provide minimum power reduction (most responsive).
/ADV:REG	Provide standard power reduction.
/ADV:MAX	Provide maximum power reduction (least responsive).
/STD	Provide standard power reduction.
/OFF	Turn power management off.

The following example shows power being installed at boot time from CONFIG.SYS. The COM ports are not monitored and the disk inactive time is set to 25 seconds.

```
INSTALL = POWER /C0 /D25
```

Note: To prevent a particular device from being monitored by POWER, enter a zero value for the inactive time. This feature enables an application to access a device directly without the possibility of the device being powered-down as the application was about to use it.

Systems Without APM

A system which is not equipped with APM BIOS can still perform its own version of power management, without using POWER. The only hardware requirement is a battery-backed real-time clock.

This can be implemented by placing the CPU in a static or halt state when the system is not in use. This is done by executing a STOP or HLT instruction when requested. When the next interrupt occurs, system operation resumes. At this time, the real-time clock is read and the BIOS tick count set, so that both of these time bases are in sync.

Non Standard Platforms/Pen Based Systems

The Power Management software was written with a standard Palm-Top PC in mind. The power management software detects system-idle when no keys are being pressed and there is no other activity in the background.

If a platform uses a pen-based system without a keyboard, the system appears to not receive user input because keys are never pressed. Such configurations require a special idle detection.

Note: Special input devices generally require more coordination between Datalight and individual OEMs. Please contact Datalight with your specific requirements.

Implementing ROM-DOS SuperBoot

Dual-booting a System Using Hidden Files

Most disk-based computer systems boot up with their primary operating system, such as Windows 95, Windows 98, Windows NT, LINUX, and others for access to the available disk drives and to establish the system environment. In some systems, it may be beneficial to employ a special boot-up mode to a different environment and a secondary operating system. A secondary operating system can be used to run special diagnostic programs that need to be kept hidden from the end-

user. Such diagnostics, accessed by a hot-key combination, may be used by service personnel or by the end user under the direction of technical support personnel.

The need to run the computer under a secondary operating system or under its primary operating system, can be met by selectively booting the computer to either of two disk drives/disk partitions included in the system. The ability to dual-boot the system is provided by a special version of Datalight's ROM-DOS that includes SuperBoot capability. A dual-boot arrangement may also be needed when it is necessary to run disk-recovery utilities or applications that are specific to the respective primary or secondary operating systems.

The remainder of this section describes the procedure for installing ROM-DOS as the secondary operating system on the system hard disk designated as the boot disk drive.

About the Boot Disk

Implementation of SuperBoot must be accomplished on the hard disk designated as the boot disk (that is, the first hard disk in the system). The secondary operating system (ROM-DOS) must be installed prior to the primary operating system and the boot disk drive must be new, or, one that can be reformatted without regard for the data which it contains. If partitions already exist on the drive, they must be removed prior to implementing the SuperBoot partition if there is insufficient free non-partitioned space on the drive to accommodate the new partition. For example, if you already have an NT partition on your disk, you can still add a SuperBoot partition if there is non-partitioned space on the first hard-drive (BIOS drive 80h). Prior to implementing SuperBoot, the hard disk must contain a low-level format as normally required before running FDISK, the partitioning utility included with ROM-DOS.

Implementation Procedure

The following procedure illustrates how to construct a SuperBoot partition and two standard ROM-DOS partitions on a hard disk. It assumes that you have installed the ROM-DOS SDK and are able to produce a standard bootable floppy disk. In addition to the development machine on which ROM-DOS is installed, you will need a second machine with a hard drive on which you can run FDISK and FORMAT, and also two floppy disks.

This example procedure creates three partitions; one SuperBoot partition and two DOS partitions. Although two DOS partitions are not required for SuperBoot implementation, the two DOS partitions demonstrate the change in drive ordering when booting using different SuperBoot libraries as a starting point. This example only proceeds through the steps using one of the SuperBoot library options. The procedure can be repeated with other library choices.

1. **Prepare a standard ROM-DOS bootable floppy disk.**
   On your development system, prepare a standard ROM-DOS bootable floppy disk. Copy the ROM-DOS FDISK and FORMAT utilities onto this floppy disk.
2. **Prepare a SuperBoot bootable floppy disk.**
   This step varies depending on the revision of ROM-DOS you are using.

   ROM-DOS 6.22, revision 3.00.1 or earlier:

If you have an existing copy of USER.LIB in the root of your ROM-DOS directory tree, rename it to USER.BAK (or other convenient name) before continuing with this procedure.

Switch to the \DATALGHT\ROMDOS\SUPRBOOT directory and issue the command:

COPY LASTB.LIB ..\USER.LIB

Once the copy is complete, change back to the root of the ROM-DOS directory and run BUILD, selecting the quick build option (as outlined in 'Chapter 4, Building ROM-DOS'). BUILD will produce a special SuperBoot version of ROM-DOS that can be used to produce a bootable floppy disk. Format the second floppy disk using the newly created ROM-DOS.SYS file and then copy the ROM-DOS FORMAT utility onto this bootable SuperBoot floppy disk.

ROM-DOS 6.22, revision 3.00.2 or later:

- Run the BUILD utility (as outlined in 'Chapter 4, Building ROM-DOS').
- Enter C in response to the "Do you wish to Quick-Build or Custom-Build ROM-DOS (Q/C)?" prompt.
- Enter Y in response to the "Will ROM-DOS boot from Floppy/Hard disk?" prompt.
- Enter Y in response to the "Would you like to enable SuperBoot support?" prompt.
- Enter L (for Last) in response to the "Specify the location for the SuperBoot drive ordering."
- Enter SuperBoot partition ID in response to the "What is the SuperBoot partition ID (in hex)?" prompt. Note the ID value you enter; this value is used in the following step.
- Respond to the remaining BUILD session prompts as they pertain to your system.

ROM-DOS 6.22, revision 3.00.2 and later, does not include a SUPRBOOT directory. Please refer to the release READ.ME file for additional information.

3. **Partition the hard drive.**
   Reboot the test machine from the standard ROM-DOS boot disk. From the DOS command prompt, issue the following commands:

```
FDISK 80 /I98 /S5 /C
FDISK 80 /B /S15 /C
FDISK 80 /B /C
```

   This example assumes the hard drive has no existing partitions and these commands do not work if the drive is already fully-partitioned. If this is the case, use the FDISK menu interface to remove all of the partitions on the disk before proceeding. These commands create a 5MB SuperBoot partition, a second 15MB partition, and a third partition that includes the remainder of the disk (or stops at the 2GB partition size limit of DOS). The value for the "I" option must match the value selected for the SuperBoot ID during the BUILD session. The default value for the SuperBoot ID is 98h.

4. **Reboot from the standard ROM-DOS floppy disk and format the bootable DOS partition.**
   Reboot from the standard ROM-DOS boot disk and then issue the following DOS commands:

```
FORMAT C: /s
FORMAT D:
```

These commands prepare the second (15MB) partition as a bootable standard DOS partition using the standard version of ROM-DOS and make the third partition a standard non-bootable DOS partition. The SuperBoot partition will not be visible to the standard ROM-DOS version nor to other operating systems.

5. **Reboot from the SuperBoot floppy disk and format the SuperBoot partition.**
Reboot the test machine using the SuperBoot version of ROM-DOS constructed in step 2.

   The 5MB SuperBoot partition appears as drive E while the 15MB standard DOS partition (made bootable in the previous step) is drive C. The third partition is drive D. The order in which DOS lists partitions is (with this particular version of the SuperBoot kernel):

   1. DOS bootable partitions
   2. DOS primary non-bootable partitions
   3. DOS extended partitions
   4. Superboot partisions

   If a different library from the SUPRBOOT sub-directory is chosen, then the relative order of the SuperBoot partition may have been different. The implications of selecting other SUPRBOOT libraries with which to build ROM-DOS are discussed later. For now, prepare the SuperBoot partition for use and continue. To complete this step, run FORMAT from the SuperBoot bootable floppy disk as shown below:

   ```
   FORMAT E: /s
   ```

6. **Reboot and activate the SuperBoot partition.**
Remove the SuperBoot floppy disk and reboot the test machine. When the lights on the keyboard flash, immediately press Alt-F2 to activate the SuperBoot partition code. A one-second delay in the boot sequence is provided for pressing the Alt-F2 keys. When the system has booted to the COMMAND prompt, issue the DIR command to verify that it has booted from the 5MB SuperBoot partition, and also note that the default drive is drive E. View a directory of drives C and D and notice that their drive order has not changed and that drive C is still the bootable DOS partition.

   While the SuperBoot kernel is active, you can issue the VER command with the /R option to display the SuperBoot options. Display of the SuperBoot options are not available when booting from the standard ROM-DOS kernel.

7. **Reboot to the standard DOS partition.**
Reboot again, but this time allow the boot process to continue without activating the SuperBoot partition using the Alt-F2 hotkey. When the boot sequence is complete, view a directory of drives C and D and verify that their order is preserved. Note that the SuperBoot partition is no longer visible and that the VER /R command no longer lists the ROM-DOS SuperBoot options.

SuperBoot Partition Order

As previously mentioned, DOS will list the drives according to their partition type. The order in which ROM-DOS presents these drives depends on whether you boot from a standard DOS kernel or a SuperBoot kernel. A standard ROM-DOS kernel will assign drive letters to partitions in the following order:

1. DOS bootable partitions

2. DOS primary non-bootable partitions
3. DOS extended partitions

ROM-DOS provides several options in selecting the relative order of a SuperBoot partition in the list above. These options are:

FIRST – The SuperBoot partition will be assigned a drive letter before any of the other drive types (typically drive C).

MIDDLE – The SuperBoot partition will always appear after the first bootable partition (typically drive D).

LAST – The SuperBoot partition will appear as the last hard drive in the system (as illustrated in the preceding example).

For each of these drive letter assignments, except First and FirstB, you also have the option of reading CONFIG.SYS and AUTOEXEC.BAT (the boot files) from either the SuperBoot drive or the standard boot drive (typically drive C). SuperBoot library files that force the kernel to read these files from the SuperBoot drive are given a "B" suffix. Note that the ROM-DOS SuperBoot kernel is only loaded from the SuperBoot partition when pressing Alt-F2 during the boot process. The drive from which the boot files are read, once the SuperBoot kernel gets control, depends on which SuperBoot library was included in the ROM-DOS build. The First and FirstB libraries are identical and both boot from the first drive letter and read the boot files from the same drive.

The following table summarizes the effects of including the various SuperBoot libraries in the ROM-DOS build. For simplicity, it is assumed that first available hard disk drive is drive C, although ROM-DOS allows for many configuration options that might change the first available hard disk drive.

Summary of SuperBoot Libraries

SuperBoot .LIB file	SuperBoot drive letter	Boot files read from drive
FIRST	C:	C:
FIRSTB	C:	C:
MIDDLE	D:	C:
MIDDLEB	D:	D:
LAST	Last hard disk drive	C:
LASTB	Last hard disk drive	Last hard disk drive

Note: The choices of FIRST, MIDDLE, and LAST are only available with ROM-DOS 6.22, revision 3.00.1 and earlier. The FIRSTB, MIDDLEB, and LASTB are available with all versions of ROM-DOS that support the SuperBoot option..

Using Win95 or Win98 as Primary Operating System

One additional step to the process outlined above needs to be taken if Win95/98 is to be used as the primary partition operating system. After running FDISK on the drive, formatting and placing the system files on primary and SuperBoot partitions as instructed, you can install Win95/98 onto the primary partition. Create a partition large enough to accommodate the installation of the

software and provide CD-ROM drivers as necessary for loading the Win95/98 software from a CD.

When the Win95/98 installation is completed, the SuperBoot partition may no longer be accessible. To correct this, reboot from the standard ROM-DOS floppy disk prepared in step 1. Rerun FDISK using the menu method (run FDISK without command line arguments). Select "V" to view the partitions and verify that the SuperBoot partition is still present. Press Esc to return to the main FDISK menu, then run the "M" option to re-write the master boot record code. Press Esc to return to the main FDISK menu, then chose "Save and Exit." When the system reboots, press Alt-F2 to activate the SuperBoot partition.

Dynamic System Configuration

Introduction

When installing software on a system or configuring a system to accommodate its current operating environment, it may be beneficial to install only certain software. For example, when booting (or installing software) from a CD-ROM, it may be desirable to install only those software components, such as device drivers, required for the particular system instead of loading the system with unnecessary software. When installing device drivers at boot time, system resources can be maximized by installing only those drivers required for the installed hardware and omitting those for which no hardware exists.

How Does Dynamic System Configuration Work?

In a system where the exact hardware configuration is unknown, following procedure is performed at boot time to determine the need for ,and to load only the required device drivers. This procedure, which relies on the Dynamic Driver Loader program and the NEWFILE feature of Datalight's ROM-DOS, programmatically determines the need for specific device drivers and subsequently loads them from the installing media.

1. Run the Dynamic Driver Loader program. This program detects the installed hardware and writes a configuration file that reflects the detected hardware.
2. Process the configuration file stored on the RAM disk to install the needed device drivers from the installation media, such as a CD-ROM.

Note: Because the dynamic configuration process requires the creation of a configuration data file (performs disk I/O), a writeable disk must be available in the target system. If no such disk is present, such as when booting from CD-ROM or read-only memory, a temporary RAM disk must be created during the configuration process.

Using the Dynamic Driver Loader

The example of dynamic system configuration presented in this document describes a method of automatically configuring a typical x86-based system. This particular example checks the system for the presence of extended memory (XMS) and if found, installs two RAM disks above the 1MB boundary. This example used is a working example because it can be implemented in any system equipped with extended memory.

Configuration is performed by loading a small program (the Dynamic Driver Loader) during the processing of CONFIG.SYS. This program examines the system for extended memory and, when found, creates a pair of RAM disks for use by installed applications. While the following example loads drivers that address memory as disk drives, the same concept can be used to load drivers for hardware components other than extended memory.

Examining the Example CONFIG.SYS File

In a ROM-DOS system, boot-up includes the processing of the CONFIG.SYS file located in the root directory of the default drive, which may be an installation CD-ROM or floppy disk. The following statement loads the HIMEM extended memory device driver, needed to use extended memory.

```
DEVICE=HIMEM.SYS
```

The next statement loads the VDISK.SYS RAM-disk driver. The Dynamic Driver example requires the use of a RAM disk on which to place the new configuration file. A customized Dynamic Driver could place the new file onto an alternate read/write drive. A 16KB RAM disk will be created on conventional memory.

```
DEVICE=VDISK.SYS 16
```

The next statement in CONFIG.SYS loads the Dynamic Driver Loader program, DYNDRVR.SYS.

```
DEVICE=DYNDRVR.SYS
```

When loaded, DYNDRVR.SYS examines the target system and detects the hardware components. DYNDRVR.SYS then lists the required device driver for any hardware it detects as a DEVICE= statement in the file DYNCFG.SYS file. In its default form, DYNDRVR.SYS detects extended memory and writes the DEVICE=VDISK statements to DYNCFG.SYS.

```
NEWFILE=A:\DYNCFG.SYS
```

The last statement in this CONFIG.SYS file uses the NEWFILE command to access the DEVICE= statements specified by the DYNDRVR.SYS program and stored in the DYNCFG.SYS file on the disk drive. In this example, the DEVICE= statements in DYNCFG.SYS establish a pair of RAM disks if extended memory is available in the system.

For the example DYNDRVR.SYS driver to work, the NEWFILE= statement must be placed in the CONFIG.SYS file exactly as shown above. The statement must be uppercase. The reference to the A drive is adjusted by DYNDRVR.SYS to reflect the correct RAM driver letter.

About the Dynamic Driver Loader

The Dynamic Driver Loader is provided in C source code and can be adjusted to accommodate a wide range of hardware. When run (during the processing of CONFIG.SYS) in its default form, DYNDRVR.SYS detects extended memory and, if present, creates a pair of RAM disks. If extended memory is not present in the target system, DYNDRVR.SYS reports an error and terminates. The basic steps taken by DYNDRVR.SYS, as provided, include:

- Examines the target hardware for the existence of extended memory (XMS).
- If XMS is found, determines which drive to write the DYNCFG.SYS file.
- Updates the NEWFILE command in CONFIG.SYS to point to the DYNCFG.SYS file.

- Creates the DYNCFG.SYS file with the DEVICE= statements needed establish the two RAM disks.

The source code for DYNDRVR.SYS is a template that can be altered and added to as needed to suit a particular target system. The source code files are located in the MEMDISK subdirectory. This driver is able to any number and type of hardware components and then insert the appropriate DEVICE= statement(s) into DYNCFG.SYS. When DYNDRVR.SYS terminates, processing of CONFIG.SYS continues with the NEWFILE command, described below. Refer to '**Using a Custom Memory Disk**' and the MAKEFILE in the MEMDISK directory for instructions on compiling the source code. To create the driver using the supplied MAKEFILE, use the command:

```
Make dyndrvr.sys
```

To manually compile the code:

```
TASM /z /mx /zd /I /DROMDISK=1 memdisk.asm
BCC -c -w -O -Z -I dyndrvr.c
     TLINK /s /m /c /l memdisk+dyndrvr,dyndrvr.sys,,memutil.lib/m;
```

About Config.sys Processing and the NEWFILE Command

The CONFIG.SYS file is loaded and interpreted by the ROM-DOS kernel. If the ROM-DOS boot diagnostics are enabled, the first portion of this processing happens after diagnostic "6".

For several reasons it is not practical to process CONFIG.SYS in the order that it appears. Among these are:

- Menu processing
- The ability to load DOS and certain device drivers into High memory
- A NEWFILE command referring to an existing drive

The alternative is a multiple-pass system, which is the way MS-DOS also chose to handle CONFIG.SYS processing. Some of this is documented in "DOS Internals" by Geoff Chappell (pp. 145-155).

Note also that F5, F8, and the SHIFT key affect processing at this level. With F8 and step-by-step confirmation, it is possible to see what we are calling PASS 2, PASS 3, and PASS 4 through the CONFIG.SYS.

In PASS 1, CONFIG.SYS is scanned for menus and blocks. If MENUs (and SUBMENUs) are present, they are displayed to the user as indicated. This also means commands related to menus are processed at this time, including MENUDEFAULT, MENUCOLOR, and NUMLOCK.

All of the menu processing results in a block being selected by the user. If there is no menu processing, the block field is essentially blank. In PASS 2, 3, and 4, the only CONFIG.SYS commands that will be processed are located:

- Before any block definitions in the file
- Within a block whose name matches the selected block
- Within a block whose name is [COMMON]

- Within any block that is specified after an INCLUDE=

In order to ease future processing and handle INCLUDE variances, ROM-DOS reprocesses these commands, in order, to a new buffer. It is at this time that NEWFILE commands will insert the newfile into this buffer.

Here is a simple example that demonstrates this behavior:

```
A:\CONFIG.SYS                  C:\CONFIG2.SYS
-------------                  --------------
DEVICE=A                       DEVICE=Z
[ONE]
DEVICE=B
NEWFILE=C:\CONFIG2.SYS
DEVICE=C
[TWO]
DEVICE=D
[COMMON]
DEVICE=E
INCLUDE=ONE
DEVICE=F
```

With this example, the devices will load in this order:

A (first in file, outside any blocks)

E (COMMON is the first block processed)

B (the ONE block is included, and processed next)

Z (bring in the newfile)

(note- ROM-DOS does NOT process the DEVICE=C)

F (back to the COMMON block)

There are other issues related to potential looping INCLUDE= blocks, but those are not within the scope of this document.

Next CONFIG.SYS Processing then continues with PASS 2. The following commands are understood:

DOS

STACKS

ROM-DOS then allocates stacks and high memory as appropriate, and continues with PASS 3. The following commands are understood:

DEVICE

DEVICEHIGH

COUNTRY

BREAK

BUFFERS

FCBS

FILES

LASTDRIVE

NUMLOCK

SHELL

SET

SEARCHES

The specified buffers, files, and FCBs are all allocated into high or low memory, and other initializations take place. If boot diagnostics are enabled, ROM-DOS emits a diagnostic "7". Finally, PASS 4 of CONFIG.SYS processing happens, and the following commands are understood:

INSTALL

INSTALLHIGH

The memory used by CONFIG.SYS and the new buffer are discarded, and the memory used by the processing code and warning messages is also discarded. If boot diagnostics are enabled, ROM-DOS emits an "8".

At this time, final processing is done and eventually the AUTOEXEC.BAT file is processed.

The NEWFILE command was originally handled during PASS 3. This was useful to our customers because they could load a device driver which created a virtual "drive" then, using NEWFILE, pass control a configuration file located on that drive.

In order to load a new CONFIG.SYS file from a virtual drive and enable DOS=HIGH commands from within the new CONFIG.SYS, the NEWFILE command was extended. The presence of an optional parameter will tell the ROM-DOS kernel to load this device driver before trying to process this NEWFILE. An example:

```
NEWFILE=C:\CONFIG2.SYS, ROMDRIVE.SYS C000
```

The only PASS2 command allowed in this variety of NEWFILE is the DOS= command, and even that is now being processed during PASS3. Any STACKS lines and options will not be processed, and must be in the base CONFIG.SYS on the boot media.

The original NEWFILE command would report an error if the target file did not exist, then continue processing in the same file. Because we will not know if the file exists until later, this new style NEWFILE command will set a flag so that no further commands will be processed from the current CONFIG.SYS file, regardless.

Building Sockets

Building the SOCKETS kernel is accomplished with a utility called SBUILD. SBUILD allows you to choose the exact type of support needed for your Sockets installation. Choices include the processor support, various network protocols and printer support.

SBUILD is an interactive program, prompting for answers to a series of questions. SBUILD requires the Borland 5.2 tools provided with the Datalight SDTK. The environment variable, BCROOT, must be set along with the path for your compiler tools. BCROOT should point to the root directory for your compiler tools. For example, if your tools are in the C:\DL\DEVTOOLS directory, you would add the following to your autoexec.bat file:

```
set BCROOT=C:\DL\DEVTOOLS
```

SBUILD performs the following operations:

- Compiles the SOCKETS configuration modules
- Links the configuration modules and libraries

SBUILD creates the file SOCKETS.EXE. Please note, your installation may include pre-built versions using the names Socketp.exe, Socketm.exe, Socketc.exe or other variations. The program SOCKETS.EXE will use the same command line arguments as Socketp and Socketm as defined in the User's Guide in the section **SOCKETS command line options.**

SBUILD Command Line Options

Ordinarily, SBUILD will be run without any command-line options. SBUILD will determine the appropriate display colors and find the assembler and linker. The following command line options are provided to correct certain error conditions.

/?	Provides online help screen.
/N	Causes SBUILD to use monochrome. Non-color displays that appear to be color displays to SBUILD, such as LCD displays, may not be readable in full color.
/P	Causes SBUILD to pause after running each sub-program. This option allows you to observe what command-lines BUILD is passing to the compiler and linker.
/T	Causes SBUILD to display in TTY mode rather than graphics. This option is necessary for incompatible monitor types.

SBUILD can rerun a session using a configuration file. Each time SBUILD runs, it saves a list of your keystrokes in a file named SBUILD.CFG. This file can be used, through a standard DOS pipe into SBUILD, to repeat the last session. For example:

```
C:\>SBUILD < SBUILD.CFG
```

If a number of standard sessions are planned, copy the file SBUILD.CFG to some other name. Then redirect that filename into SBUILD any number of times. SBUILD also creates a file named SBUILD.TXT. This file contains a complete list of the questions and the answers you selected during the last SBUILD session and is the same information as on SBUILD's final confirmation screen. SBUILD.TXT can be referenced when calling technical support or saved with your project for future reference.

The third output file from SBUILD.EXE is SBUILD.BAT. SBUILD.BAT together with COMPILE.CL and SOCKETS.LNK contain a complete set of instructions for compiling and linking the version of SOCKETS that was set up in the previous run of SBUILD. Executing SBULD.BAT generates a copy of the previous SOCKETS kernel without running the SBUILD program. SBUILD.BAT relies on the existence of the files COMPILE.CL and SOCKETS.LNK (linking command line). These files are generated during the SBUILD session.

Note: To run SBUILD.BAT, you must specify the .BAT extension, otherwise the .EXE extension is assumed and SBUILD.EXE runs.

Datalight recommends saving a copy of SBUILD.BAT, COMPILE.CL, SOCKETS.LNK and SBUILD.TXT under different names or in a separate directory when you successfully create a working Sockets kernel. This ensures that you can always re-create the same working SOCKETS kernel configured for your exact needs.

Note: Each revision of SBUILD may change; do not use old configuration files with a new SBUILD.

If you want to change the default colors, specify the new colors in a text file named SBUILD.COL. The colors must be listed as four comma-separated integers, on the first line of the file. The numbers represent the background, window, error, and question colors, using the standard color mapping. For example, to set a gray background with white text, a blue text window with white text, a red error window with white text, and a blue question prompt with yellow text, enter:

```
C:\> COPY CON SBUILD.COL 127, 31, 79, 30 <Ctrl-Z>
```

Warning: Do not change the SOCKETS configuration source files CONFIG.C and CONFIGR.C in any way. Doing so may interfere with the operation of SBUILD.

Before Running SBUILD

Before you run SBUILD, you will need to be prepared to make several decisions based on your hardware, your network configuration, and your network application and communication needs. SBUILD provides many options for configuring your SOCKETS kernel.

- **Is your processor an Intel 386 or better?** If not, you can build a version of SOCKETS that will run on your 186 or 286 system.
- **Do you want IPv4 or IPv6 support or both?** Most systems today support the widely used network protocol IP version 4, but some newer systems are supporting the Next Generation network protocol IP version 6
- **Will your system use a Network Interface Card or Serial Communication, either with a direct connection or a modem for communication? Will it use more than one method?** Sockets will need to support the Packet Driver interface for systems using Network Interface cards, and direct serial communication for directly connected asynchronous serial lines or modems. SOCKETS can be configured to support both methods, or only one, as needed.
- **What network protocols and information and support do you need?** SOCKETS can be configured to keep MIB II statistics, support Routing Information Protocol (RIP) for IPv4, support a print server and client, provide alternative interfaces for destinations (proprietary protocol for IPv4 only) and to make use of the Internet Gateway Management Protocol (IGMP) for configurations containing a Network Interface Card with IPv4. For Serial Communication SLIP, Compressed SLIP and PPP can be configured for IPv4, but PPP only is configured automatically for IPv6. For PPP the optional CHAP-MD5 authentication protocol can be configured and for SLIP and CSLIP

the optional proprietary Modem Pool support. Modem Pool support is also known as Multi Destination Drivers.

SBuild Sample Sessions

SBUILD allows you to create a variety of different Sockets kernels. You must have the compiler and linker tools in your path (Borland's BCC and TLINK) for SBUILD to complete it's process. These tools are available in the Developer's Toolkit. If SBUILD does not find an available compiler and linker, it warns you and gives you an option to proceed anyway or quit the SBUILD process. You can press Esc to exit SBUILD at any time. Several examples are shown below. The output from SBUILD is shown in block letters. The user-entered responses are shown in bold.

Example 1:

If your system will make use of both IPv4 and IPv6 support and Ethernet connectivity, keep MIB II statistics and provide Printer support, you would answer the SBUILD prompts as follows using the **C**ustom build option:

Would you like to build for Intel386 or better? **Y**

Would you like IPv4 support? **Y**

Would you like IPv6 support? **Y**

Would you like Packet Driver support? **Y**

Would you like Serial Communication Port support? **N**

Would you like Sockets to keep MIB II statistics? **Y**

Would you like RIP support? **N**

Would you like Printer Support? **Y**

Would you like Alternative Interface support? **N**

Would you like IGMP support? **N**

Example 2:

If your system will use Serial Communication Port support, does not require additional networking protocols, and will not use Printer server and client support, you can follow the **Q**uick build path:

Would you like to build for Intel386 or better? **Y**

Would you like IPv4 support? **Y**

Would you like IPv6 support? **N**

Would you like Packet Driver support? **N**

Note: Serial Communication Port support is assumed if you select No for Packet Driver support.

Using **Q**uick build, the default options for all additional network protocols and statistics abilities are used. In this case, that would mean the Sockets kernel would not keep MIB II statistics, would not support Routing Information protocol, Alternative Interfaces, and IGMP support, and would include Printer support. PPP and Modem would be supported, but not SLIP, CSLIP, CHAP-MD5 authentication and Modem pool.

SOCKETS Programming Tutorial

Sample Programs

Compiler Notes

The attached examples are designed for use with the Borland C++ 5.2 or the Microsoft VC++ 1.5 compilers. Makefiles (example.mak and example.ms) are provided for reference.

The module "compiler.h" can be ported for use with other compilers, currently it supports various Microsoft and Borland compilers and the DJGPP compiler for 32 bit DOS with a DPMI DOS extender.

To use the makefile with BC 5.2 simply type:

```
make –fexample.mak
```

Included Files

- CHAT.C
- MCCHAT.C
- SCHAT.C
- CHAT.IDE
- MCCHAT.IDE
- SCHAT.IDE
- EXAMPLE.MAK
- EXAMPLE.MS
- CAPI.C
- CAPI.H
- COMPILER.H
- _CAPI.C
- CAPIS.LIB
- SOCKETS.LIB

CHAT

Overview

A TCP based CHAT application. A server is started on the defined CHAT port. All connections made to this server, as well as those made by the local user to other servers, are put in a list. Whatever data the local user enters is sent to all the connections in this list (When the user hits Enter). Any data received from any of these listed connections is displayed on the screen.

This program and its' functions are single-threaded and non-reentrant and should be used as such.

Protocol

A CHAT server accepts TCP connections from several clients on port 5000. Once connected, a client may send lines of text to the server. Those lines are then sent out to all connected clients. The net result is that all connected users can see the typed lines of text from all other users.

Implementation

For the sake of simplicity, this implementation is not designed for portability. Since the Compatible API is only available for DOS-based stacks, the code relies on certain DOS features like keyboard hardware. Also, several basic functions are simply presented rather than explained.

Programming Style and Naming Conventions

Datalight strongly recommends the use of the Hungarian naming convention. The code in this tutorial relies on that convention. For those unfamiliar, Datalight recommends several reads of the Microsoft Press book, "Writing Solid Code" by Steve Maguire. Here are a few of the prefixes and a brief explanation:

i	integer
sz	string, terminated by zero
rg	range, an array of elements
c	character
p	pointer

CAPI Variable and Functions Used

- iNetErrNo
- GetAddressInfo
- SetSocketOption
- GetSocket
- ListenAcceptSocket
- AcceptSocket
- ConnectSocket
- WriteSocket
- ReadSocket
- ReleaseSocket

Includes and Defines

These includes and defines are needed by later code pieces.

```
#define IPV6            // Compile for both IPv4 and IPv6
```

```
#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
#include <process.h>
#include "compiler.h"
#include "capi.h"

// Prototypes
int ConnectTo(char* pcHostName);
int StartListen(int iPort);
char * GetErrorString(unsigned uErrCode);
void Aprintf (char *pFormat, ...);
static char *WriteName(NET_ADDR *psAddr,int iAdLen);
static void ResetEnum(void);
static int GetNextIndex(void);
static int GetFirstOpen(void);

#define CHAT_PORT 5000
#define MAX_CONS 30
#define BUF_SIZE 200

static int rgiSocks[MAX_CONS];        // all the sockets (passive and active)
static char rgszNames[MAX_CONS][80]; // Names associated with connections
static char rgcConnecting[MAX_CONS]; // Set if starting a connection
static NET_ADDR sNetAddr;             // for general use

static char rgcKeyBuf[BUF_SIZE];      // key input buffer
static int iKeyCount = 0;             // number of characters in above buffer
WORD wChatPort = CHAT_PORT;
```

Utility Functions

The following section of code shows three useful utility functions. Brief comments before each function should provide explanations.

```
/*
   Create a human understandable string from a SOCKETS error code.
   Argument:
       uErrCode - The SOCKETS error code.
   Returns:
       A pointer to the (static) string representation of the error.
*/
char *GetErrorString(unsigned uErrCode)
{
    static char rgcUnk[30];
    static char *rgszErrs[] =
    {
        "NoErr",
        "InUse",
        "DOSErr",
        "NoMem",
        "NotNetconn",
        "IllegalOp",
        "BadPkt",
        "NoHost",
        "CantOpen",
        "NetUnreachable",
        "HostUnreachable",
        "PortUnreachable",
        "PortUnreachable",
```

```
        "TimeOut",
        "HostUnknown",
        "NoServers",
        "ServerErr",
        "BadFormat",
        "BadArg",
        "EOF",
        "Reset",
        "WouldBlock",
        "UnBound",
        "NoDesc",
        "BadSysCall",
        "CantBroadcast",
        "NotEstab",
        "ReEntry",
        "Network",
        "Terminating",
        "InfoLocked",
        "BadInterface",
    };

    if (uErrCode == ERR_API_NOT_LOADED)
        return "Sockets not loaded";
    if ((uErrCode & 0xff) > ERR_BAD_INTERFACE)
    {
        sprintf(rgcUnk,"Unknown error 0x%04X",uErrCode);
        return rgcUnk;
    }
    return rgszErrs[uErrCode & 0xff];
}


/*
   Show data on screen.  For now we just use vprintf to print the data and
   reprint any stuff that was being edited.  Accepts variable number of
   parameters, exactly like printf, and process them using the va_list/va_args
   method.
   Arguments:
       pFormat and ellipsis - Exactly the same as for printf().
*/
void Aprintf (char *pFormat, ...)
{
    va_list pArgs;

    if (iKeyCount)        // data been edited, start on newline
        printf("\n");

    va_start(pArgs, pFormat);
    vprintf(pFormat, pArgs);
    va_end(pArgs);

    // print the old edited stuff (if any)
    if (iKeyCount)
    {
        rgcKeyBuf[iKeyCount] = 0;
        printf("%s", rgcKeyBuf);
    }
}


/*
   Create a string representation of the IP address and port, in the form
```

```
   [a.b.c.d]:port, e.g. [196.10.180.3]:1400.
    or
    [a::b:c:d:e]:port, e.g. [3000::Co:1234:ABCD:12AB]:1400
   Arguments:
       psAddr - pointer to NET_ADDR structure containing address of host.
   Returns:
       Pointer to (static) array containing null-terminated string.
*/
char *IptoAsc(IPAD *psIpAddress,int iAdLen);
static char *WriteName(NET_ADDR *psAddr,int iAdLen)
{
    static char rgcName[60];

    sprintf(rgcName, "[%s]:%u",IptoAsc(&psAddr->sIpAddr,iAdLen),
                               psAddr->wRemotePort);
    return rgcName;
}

// utility function to reverse a 16 byte IPv6 address
BYTE *Put128(BYTE *pbDest,BYTE *pbSource)
{
   int i;

   pbSource += 15;
   for (i = 0; i < 16;++i)
      *pbDest++ = *pbSource--;
   return pbDest;
}

#define get16(p) ((WORD)((((WORD)p[0]) << 8) | p[1]))

// Convert an IPv4 or IPv6 address to printable format
char *IptoAsc(IPAD *psIpAddress,int iAdLen)
{
   static char sz[42];
   char *psz = sz;
   BYTE *pb;
   int i,iZero,iZeroCurrent = -1,iNumZeros,iNumZerosCurrent;
   WORD wVal;
   BYTE abIpAddress[16];

   Put128(abIpAddress,psIpAddress->abAddr);
   if (iAdLen == 4) {
      sprintf(sz,"%u.%u.%u.%u",
         psIpAddress->abAddr[3],
         psIpAddress->abAddr[2],
         psIpAddress->abAddr[1],
         psIpAddress->abAddr[0]);
      return sz;
   }
   for (i = iZero = iNumZeros = iNumZerosCurrent = 0,
                   pb = abIpAddress;i < 17;i += 2,pb += 2) {
      if (i < 16 && *(WORD *)pb == 0) {
         iNumZerosCurrent += 2;
         if (iZeroCurrent < 0)
            iZeroCurrent = i;
      }
```

```
            else {
               if (iNumZerosCurrent > iNumZeros) {
                  iNumZeros = iNumZerosCurrent;
                  iZero = iZeroCurrent;
               }
               iNumZerosCurrent = 0;
               iZeroCurrent = -1;
            }
            if (i == 16)
               break;
         }
         for (i = 0,pb = abIpAddress;i < 16;i += 2,pb += 2) {
            if ((wVal = get16(pb)) != 0 || i <= iZero ||
                                       i > iZero + iNumZeros) {
               if (!wVal && i == iZero) {
                  *psz++ = ':';
                  if (!i)
                     *psz++ = ':';
               }
               else
                  psz += sprintf(psz,"%X%s",wVal,i == 14 ? "" : ":");
            }
         }
         *psz = 0;
         return sz;
      }
```

Array Maintenance Functions

The uses of these functions are not quite as obvious as the last set. They operate on an array of socket handles. If an array element is zero, then it is not allocated. Otherwise, it is an open socket that may have data pending for send or receive. One useful function is to find the first unallocated entry so that it can be allocated. Another would be to find the next allocated entry to test its data-pending state. The following functions implement all of the code needed to perform these routines.

```
/*
   Gets the first not-assigned entry in the rgiSocks array.
   Returns:
       The first nonzero index or -1 if all in use.
*/
static int GetFirstOpen(void)
{
    int iRet = 0;
    while (iRet < MAX_CONS)
    {
        if (rgiSocks[iRet] == 0)
            return iRet;
        iRet++;
    }
    return -1;
}

/*
   The next two functions implements an enumerator. To initialize (or reset),
   call resetEnum. Successive calls to getNextIndex returns all the used
   indexes. When there is no more used indexes, -1 is returned
```

```
*/
static int iPrev = -1;     // privately used by next two functions
static void ResetEnum(void)
{
    iPrev = -1;
}

/*
   Get the next not 0 entry in the rgiSocks table.
   Returns:
       The index of the next entry or -1 if no more.
*/
static int GetNextIndex(void)
{
    while (++iPrev < MAX_CONS)
    {
        if (rgiSocks[iPrev] != 0)
            return iPrev;
    }
    return -1;
}
```

Connection Functions

These functions are simple wrappers that isolate common, redundant code for easier debugging and use. They are so common that they can be copied verbatim into your application if you like.

```
/*
   Open a TCP connection to the specified host and the default CHAT port.
   Argument:
       pHostName - A pointer to the name of the host we want to connect to
   Returns:
       A descriptor of the newly created socket.
*/
int ConnectTo(char* pcHostName)
{
    int iSock;      //descriptor

    memset(&sNetAddr, 0, sizeof(NET_ADDR));

    sNetAddr.wRemotePort = wChatPort;

    // execute several commands and test for error at each step
    if ((iAdLen = GetAddressInfo(pcHostName,
            AI_PARSE | AI_HOSTTAB | DNS_IPV4 | DNS_IPV6 , &sNetAddr)) == 0)
      Aprintf("Error on GetAddressInfo: %s\n", GetErrorString (iNetErrNo));
    else if ((iSock = GetSocket()) < 0)
    {
        Aprintf("Error on GetSocket(): %s\n", GetErrorString(INetErrNo));
    }
    else if (SetSocketOption(iSock, 0, NET_OPT_NON_BLOCKING, 1, 1) < 0)
    {
        Aprintf("Error on SetSocketOption(): %s\n",
                                        GetErrorString(INetErrNo));
    }
    else if (ConnectSocket(iSock, STREAM, &sNetAddr) < 0)
    {
        Aprintf("Error on ConnectSocket: %s\n", GetErrorString(INetErrNo));
    }
    else
    {
```

```
        Aprintf("Trying to connect to %s (%s)\n",pcHostName,
                                            WriteName(&sNetAddr,iAdLen));
        return iSock;
    }

    Aprintf("Error on connect to %s - no con\n", pcHostName);
    return 0;
}

/*
   Start a TCP server on specified port. (Returns immediately)
   Argument:
       iPort - The TCP port number to listen on.
   Returns:
       A descriptor of the server socket, or 0 if an error occured.
*/
int StartListen(int iPort)
{
    int iSock;

    memset(&sNetAddr, 0, sizeof(NET_ADDR));
    sNetAddr.wLocalPort = iPort;

    if ((iSock = GetSocket()) < 0)
    {
        Aprintf("Error on serv GetSocket(): %s\n",
                GetErrorString(INetErrNo));
    }
    else if (SetSocketOption(iSock, 0, NET_OPT_NON_BLOCKING, 1, 1) < 0)
    {
        Aprintf("Error on serv setOpt(): %s\n", GetErrorString(INetErrNo));
    }
    else if (ListenAcceptSocket(iSock, STREAM, 5, &sNetAddr) < 0)
    {
        Aprintf("Error on serv net_listen: %s\n",
                GetErrorString(INetErrNo));
    }
    else
    {
        return iSock;
    }
    return 0;
}
```

CHAT Code Loop

The following is the main code loop for the CHAT program. It relies on all the functions above, and builds on top of them using a common socket-polling methodology.

```
/*
   Loop and look for both user input or network input.
*/
void main(void)
{
    // listening socket, copied to rgiSocks once connection is made
    int iListenSock;
    int iSock;

    // general variables
    char rgcBuf[BUF_SIZE];
    int iIndex, iLength;
```

```
char cCh;

// clear all client sockets
for (iIndex = 0; iIndex < MAX_CONS; iIndex++)
    rgiSocks[iIndex] = 0;

// start the server
iListenSock = StartListen(CHAT_PORT);

// give a visual cue for users
Aprintf("Press Alt-H for help\n");

// loop forever, looking for network/keyboard input
while (1)
{
    // server part - see if new connection was opened
    iSock = AcceptSocket(iListenSock, STREAM | TYPE_EXT, &sNetAddr);
    if (iSock >= 0)
    {
        // client is connected, allocate array entry
        if ((iIndex = GetFirstOpen()) != -1)
        {
            // record this as a client connection
            rgiSocks[iIndex] = iListenSock;

            // show the connection
            sprintf( rgszNames[iIndex], "%s",
                        WriteName(&sNetAddr, (int)sNetAddr.dwRemoteHost));
            Aprintf("Connection made by %s\n", rgszNames[iIndex]);

            // hello the new client
            iLength = WriteSocket(rgiSocks[iIndex],
                                  "(server)Hello!", 14, 0);
            if (iLength < 0)
            {
                Aprintf("Error on hello: %s\n",
                        GetErrorString(INetErrNo));
            }
        }
        else
        {
            // no free slots available
            ReleaseSocket(iSock);
        }
    }
    else if (iNetErrNo != ERR_WOULD_BLOCK)
    {
        static int LastErr = 1000;
        if (iNetErrNo != LastErr)
        {
            LastErr = iNetErrNo;
            printf("Error = %d %s\n",iNetErrNo,Err(iNetErrNo));
        }
    }

    // client part - scan for data ready to recieve
    ResetEnum();
    while ((iIndex = GetNextIndex()) != -1)
    {
        // any data on this socket?
        iLength = ReadSocket(rgiSocks[iIndex], rgcBuf, BUF_SIZE, 0, 0);
        if (iLength <= 0)
```

```
    {
        if (iNetErrNo == ERR_WOULD_BLOCK ||
                (iNetErrNo == ERR_NOT_ESTAB && rgcConnecting[iIndex]))
        {
            // nothing to read or not established yet
            continue;
        }
        Aprintf("Error on ReadSocket from %s: %s - closing\n",
                    rgszNames[iIndex], GetErrorString(INetErrNo));
        // dead client, free socket
        ReleaseSocket(rgiSocks[iIndex]);
        rgiSocks[iIndex] = 0;
    }
    else
    {
        // data received from client
        // force string termination
        rgcConnecting[iIndex] = 0;
        rgcBuf[iLength] = 0;

        // show the node and string
        Aprintf("%s: %s\n", rgszNames[iIndex], rgcBuf);
    }
}

// local keyboard, were any keys hit?
if (kbhit())
{
    // yes.  which ASCII value?
    switch(cCh = getche())
    {
        // NULL -> extended key.
        case 0:
        {
            printf("\n");
            switch (cCh = getch())
            {
                // unexpected value?
                default:
                printf("Undefined function key: %d\n", cCh);
                // fall through

                // Alt+H - Help
                case 35:
                printf("Alt-C    Close connection\n"
                    "Alt-N    New connection\n"
                    "Alt-H    Help\n"
                    "Alt-L    List connections\n"
                    "Alt-X    eXit\n");
                break;

                // Alt+X - Exit
                case 45:
                goto exit;

                // Alt+N - Connect as client
                case 49:
                // make a new connection
                if ((iIndex = GetFirstOpen()) != -1)
                {
                    // get destination host
                    printf("Enter destination:");
```

```
            gets(rgcBuf);

            // attempt connection
            rgiSocks[iIndex] = ConnectTo(rgcBuf);

            // if success, copy the name
            if ((rgiSocks[iIndex]) != 0)
            {
                rgcConnecting[iIndex] = 1;
                strcpy( rgszNames[iIndex], rgcBuf );
            }
        }
        else
        {
            printf("Max number of connections in use\n");
        }
        break;

        // Alt+L - List connections
        case 38:
        printf("List of all connections\n");
        ResetEnum();
        while((iIndex=GetNextIndex()) != -1)
        {
            printf("Connection #%d descriptor:%u name %s \n",
                 iIndex, rgiSocks[iIndex], rgszNames[iIndex]);
        }
        printf("List end\n");
        break;

        // Alt+C - Close a connection
        case 46:
        printf("Enter connection to close:");
        iIndex = atoi(gets(rgcBuf));
        if (iIndex < 0 || iIndex > MAX_CONS)
        {
            printf("OUT of range:%d\n", iIndex);
        }
        else if (rgiSocks[iIndex])
        {
            ReleaseSocket(rgiSocks[iIndex] );
            rgiSocks[iIndex] = 0;
            printf("Closed %s\n", rgszNames[iIndex]);
        }
        else
        {
            printf("Connection %d not open\n", iIndex);
        }
        break;
    }

    // just for looks
    Aprintf("");
    break;

    // normal key
    default:
    if (iKeyCount < BUF_SIZE)
    {
        // we have room, store it
        rgcKeyBuf[iKeyCount++] = cCh;
        break;
```

```
                        }
                        // buffer full, force send now
                        // fall through

                        // enter - send buffer now
                        case '\r':
                        printf ("\n");
                        if (iKeyCount == 0)
                           break;

                        // write data to all connections
                        ResetEnum();
                        while ((iIndex = GetNextIndex()) != -1)
                        {
                            iLength = WriteSocket( rgiSocks[iIndex], rgcKeyBuf,
                                  iKeyCount, 0);
                            if (iLength < 0)
                            {
                                // write failed!
                                Aprintf("Error on NetWrite from %d %d bytes: %s -
                                closing connection\n", iIndex, iKeyCount,
                                GetErrorString(INetErrNo));
                                ReleaseSocket( rgiSocks[iIndex]);
                                rgiSocks[iIndex] = 0;
                             }
                        }

                        // forget everything we just wrote
                        iKeyCount = 0;
                        break;
                    }
                }
            }
        }

        // done running!
        exit:

        // release all sockets
        ResetEnum();
        while ((iIndex = GetNextIndex()) != -1)
            ReleaseSocket(rgiSocks[iIndex]);
        ReleaseSocket(iListenSock);
    }
```

MCCHAT

Please review the differences between a TCP (stream) session and a UDP (datagram) session. When designing an application to use a UDP session we eliminate many connection issues by simply not caring if the recipient has in fact received the packets being sent.

'CHAT' means to to talk and to listen. When using UDP, it means that for the talk size we just send a multicast message on the LAN and for the listen side we start a UDP server.

This program and it's functions are non-reentrant.

CAPI Variable and Functions Used

- iNetErrNo
- GetKernelInformation
- JoinGroupEx
- LeaveGroupEx
- GetAddressInfo
- SetSocketOption
- GetSocket
- ConnectSocket
- WriteSocket
- ReadSocket
- ReleaseSocket

Source code

The source code can be found as MCCHAT.C in the EXAMPLES folder and is not repeated here.

Array Maintenance Functions

Not required in MCCHAT.

SCHAT

SCHAT has the same functionality as CHAT, but uses the Sockets API instead of CAPI. The source code SCHAT.C can be compiled for either DOS using Sockets or Windows using WinSock, demonstrating the portability of the code.

Sockets Functions Used

- WSAGetLastError
- WSAStartup
- WSACleanup
- getaddrinfo
- freeaddrinfo
- socket
- ioctlsocket
- bind
- accept
- connect
- send
- recv
- closesocket

Advanced Examples

Two diagnostic programs are provided in source and binary format both as advanced examples and useful utility programs. They are CAPIDIAG which performs a comprehensive diagnostic of the Compatible API and SOCKDIAG which does the same for the Sockets API. In addition to using all the API functions they also test the Sockets stack by using both loopback (internal) echo servers and external echo servers on peers to test ICMP Ping, TCP and UDP traffic.

Index