Embedded DOS$_{tm}$ 6-XL

The Full-Featured DOS for Embedded Systems

Developer's Guide
with Command Reference

General Software, Inc.
P.O. Box 2571
Redmond, Washington 98073

Tel (206) 454-5755
FAX (206) 454-5744
BBS (206) 454-5894

Email: general@gensw.com

IMPORTANT NOTICES

Important Licensing Information

1. EMBEDDED DOS 6-XL is licensed for use as an operating system for embedded systems. It is not licensed for use as a general purpose desktop operating system where the end user has control over which third party programs are run on the target system.

2. The source, object, and executable software provided with General Software's Adaptation Kits involve valuable copyright, trade secret and other proprietary rights of General Software, Inc. No title to or ownership of the software, or the proprietary rights associated with the software, is transferred to you with this package.

3. General cannot possibly anticipate the end applications in which EMBEDDED DOS 6-XL is used, and cannot be certain that all methods used to develop software for MS-DOS are truly portable to the EMBEDDED DOS 6-XL environment. ACCORDINGLY, YOU ACCEPT THE SOFTWARE 'AS IS' AND WAIVE AND RELEASE ALL REPRESENTATIONS, WARRANTIES AND LIABILITIES OF GENERAL SOFTWARE, EXPRESS OR IMPLIED, ARISING BY LAW OR OTHERWISE, WITH RESPECT TO THE SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY: (A) IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE; (B) IMPLIED WARRANTY ARISING FROM COURSE OF PERFORMANCE, COURSE OF DEALING OR USAGE OF TRADE; (C) CLAIM OF INFRINGEMENT; OR (D) ANY OBLIGATION, LIABILITY, RIGHT, REMEDY OR CLAIM IN TORT, NOTWITHSTANDING ANY FAULT, NEGLIGENCE, STRICT LIABILITY, OR PRODUCT LIABILITY OF GENERAL (WHETHER ACTIVE, PASSIVE OR IMPUTED).

TABLE OF CONTENTS

ABOUT THIS DEVELOPMENT KIT

Thank you for choosing General Software's EMBEDDED DOS 6-XL operating system for use in your embedded system. This version of DOS offers a superior combination of embedded features, performance, and MS-DOS compatibility that will satisfy your needs for both a ROM-able DOS operating system, and even a real-time kernel.

Chapter 1

INTRODUCTION

Introduction to EMBEDDED DOS 6-XL

EMBEDDED DOS 6-XL is a high-performance, full-featured, ROM-able operating system specifically designed for use in embedded systems. In addition to providing compatibility with the majority of desktop software (such as Windows 3.1, Norton Utilities, or Brief, for example), it offers real-time multitasking functionality that comes as an integrated part of the operating system, so that real-time applications can be built using DOS tools.

EMBEDDED DOS 6-XL's full implementation of MS-DOS software interrupt functions makes it simple to develop and test application code built with standard MS-DOS development tools, and eliminates the need for special, non-DOS support libraries.

The enhanced real-time features of EMBEDDED DOS 6-XL are fully supported on the PC platform itself, allowing the designer to work with the built-in threads, lightweight timers, events, and mutex semaphores on both the development and target machines. Because EMBEDDED DOS 6-XL is compatible with your desktop machine, you can fully test the operation of your multitasking embedded code directly on a PC.

The EMBEDDED DOS 6-XL system is implemented in hand-optimized 8086 and '386 assembly language for optimal performance, full reentrancy, and minimum size. The EMBEDDED DOS 6-XL kernel is under 50 KB in size, whereas the equivalent MS-DOS kernel is 71 KB in size.

EMBEDDED DOS 6-XL is ROMable in two ways. It may be loaded from a single ROM BIOS extension or stored as a file on a ROM disk.

The EMBEDDED DOS 6-XL system comes with a kernel debugger that works closely with the EMBEDDED DOS 6-XL system to help you debug new device drivers, file system drivers, TSRs, and multithreaded application programs. The debugger loads as a device driver from CONFIG.SYS and can use the PC keyboard and screen, or redirect its output to a serial port for remote debugging.

EMBEDDED DOS 6-XL comes with two command interpreters: one that functions substantially like the MS-DOS COMMAND.COM, and a tiny COMMAND.COM that saves low memory space by running in only 12 KB for memory-limited systems. The standard COMMAND.COM features a superset of its MS-DOS counterpart's commands, and swaps itself to XMS leaving

only a 4 KB footprint when running applications. The tiny COMMAND.COM has a limited set of commands with limited operand parsing.

Standard DOS utilities are provided with the EMBEDDED DOS 6-XL operating system, including CHKDSK, DISKCOPY, DISKCOMP, FORMAT, FDISK, SMARTDRV, SYS, and others. These utilities work largely the same as their MS-DOS counterparts.

Standard DOS device drivers are also provided. They include ANSI.SYS, RAMDISK.SYS, VDISK.SYS, IFSFUNC.SYS, HIMEM.SYS, POWER.SYS, and DEBUG.SYS. Some of these drivers operate differently from their MS-DOS counterparts as necessary to work in the real-time EMBEDDED DOS 6-XL environment.

Getting Help with Related Embedded Products

General Software manufactures several related products that can assist you with your embedded design, development, or debugging efforts. Contact your General Software dealer for more information.

Related Reading

For background information, or reference use, we suggest that you read the following related publications. We use these resources for developing applications for our customers.

Brown, Ralf, PC Interrupts, Addison/Wesley.

Duncan, Ray, MS-DOS Functions, Microsoft Press.

Intel, 386 DX Microprocessor Programmer's Reference Manual #230985-003, Intel Corporation.

Microsoft Press, MS-DOS Programmer's Reference, Microsoft Corporation.

Phoenix, System BIOS for IBM PC/XT/AT Computers, Addison/Wesley.

Schulman, Andrew, Undocumented DOS, Addison/Wesley.

Chapter 2

BOOTING EMBEDDED DOS 6-XL

Running an Embedded Application

Your embedded application program may be run from the standard COMMAND.COM command interpreter, from the tiny COMMAND.COM command interpreter, or from CONFIG.SYS.

The Standard COMMAND.COM

The simplest way to run an application program on an EMBEDDED DOS 6-XL system is to run it as a loadable program from the command line, via one of the COMMAND.COM interpreters. The standard COMMAND.COM provides substantially similar functionality to its MS-DOS counterpart, and adds features beyond the MS-DOS COMMAND.COM.

Enabling/Disabling the Task Bar

On desktop systems, the standard COMMAND.COM uses the special character sequence, $i, in the PROMPT string to mean that it should display a "*task bar*" on the top line of the screen before accepting a command from the user. This task bar is primarily used to notify the user that EMBEDDED DOS 6-XL, and not generic DOS, is running on the computer.

To enable the task bar, include the $i sequence in the PROMPT string, as in the following example:

```
C> SET PROMPT=$i$p$g
C:\GENERAL>
```

To disable the task bar, leave the $i sequence out of the PROMPT string. This can be done at AUTOEXEC.BAT time to turn off the task bar early in system initialization.

To disable the taskbar from CONFIG.SYS, you can specify it when using the SHELL= command with the '/s' option:

```
SHELL=COMMAND.COM /s
```

This instructs COMMAND.COM to not do any direct screen writing since it may be used in a serial communications line application.

Additional Commands in COMMAND.COM

The standard COMMAND.COM differs from its generic DOS counterpart by providing additional commands.

Additional commands include such features as ASK (to prompt for a string from the command-line user), SYNCH (to flush all data to disk and invalidate the cache before rebooting), and REBOOT (to reboot the machine). For a complete list of commands, issue the HELP command.

The Tiny COMMAND.COM

You may wish to save ROM and RAM space with the reduced version of COMMAND.COM, the tiny command processor. This interpreter supports basic DIR, TYPE, COPY, RENAME, and other COMMAND.COM commands, but without wildcarding or extra switches. For example, a command such as `COPY *.DAT \FOO` would not be allowed.

The tiny COMMAND.COM does not provide PROMPT or PATH services, and has a primitive critical error handler that simply resets the command interpreter to a known state and reprompts for a command.

This tiny command processor does support batch file processing, with GOTO and IF commands.

The tiny version of COMMAND.COM uses about 12KB of system RAM, making it ideal for "RAM cram" application situations.

Running Without COMMAND.COM

As an alternative to running your program from the command line, you may have DOS file run your application directly, with SHELL= in CONFIG.SYS.

Running From CONFIG.SYS

To execute your program from CONFIG.SYS (thereby avoiding the loading of any version of COMMAND.COM), insert the following line in your CONFIG.SYS file:

```
SHELL=d:\path\progname.ext
```

or,

```
INSTALL=d:\path\progname.ext
```

In this example, the drive letter, path, program name, and extension (COM or EXE) are spelled out in full, although the drive and path may be omitted if the program is found in the root directory. The program name and extension are required.

Operands may follow the extension, provided they are separated by spaces. These operands will be passed to your application through the PSP's command line area.

The SHELL= or INSTALL= statements in your CONFIG.SYS file are read on the last pass, so that all other statements are processed before running your program.

Chapter 3

CONFIGURING EMBEDDED DOS 6-XL

EMBEDDED DOS 6-XL is easily configurable through commands entered in its CONFIG.SYS file. This file must reside in the root directory of the boot device in order to be effective.

General Format of the CONFIG.SYS File

The standard format for CONFIG.SYS files is a sequence of ASCII text lines, where the total size of the CONFIG.SYS file does not exceed 64KB. Blank lines, and lines starting with ';', are ignored during CONFIG.SYS processing. Each line must end with a carriage-return, a line-feed, or both. Control characters encountered on lines cause the remainder of the line to be unprocessed by EMBEDDED DOS 6-XL, except for ^I (TAB).

CONFIG.SYS is actually processed several times by EMBEDDED DOS 6-XL during system initialization. Each pass collects the proper information to initialize the system at its level. For example, all device drivers are loaded first before UMB statements can be processed, or memory managers would not be able to initialize memory before that memory was linked into the system's memory arena.

CONFIG.SYS Commands

? Command	The "?" command is used to prefix any line in CONFIG.SYS that is optional. If "?" is placed in front of any CONFIG.SYS command, EMBEDDED DOS 6-XL will display the line and prompt the user to press "Y" if the line should be executed, and "N" if it should be bypassed. This allows optional execution of CONFIG.SYS commands at the user's discretion.
BREAK=ON|OFF	The BREAK command sets the default value of the DOS break flag. The DOS break flag is used by EMBEDDED DOS 6-XL to determine if it should poll the console for a ^C from the operator when an INT 21h function is invoked. If BREAK=ON, then the console is polled, and if a ^C is pending, the program is

	terminated. If BREAK=OFF, then the console is not polled. Polling adds an extra trip through the I/O system for each INT 21h call, so performance is best with it off.
BUFFERS=*nnnn*	The BUFFERS command remains for compatibility with MS-DOS CONFIG.SYS files; it has no effect on operation of the system. EMBEDDED DOS 6-XL uses a file system cache, not a buffer system, that has different allocation strategies.
CACHESIZE=*nnnn*	The CACHESIZE command sets the ceiling on the number of 512-byte cache blocks that can be active in the system at any given time. Cache blocks are dynamically allocated from system pool, and after aging, they are automatically returned to pool.
CACHETTL=*nnnn*	The CACHETTL command sets the number of milliseconds that a cache block can remain in the system without being referenced by the file system until it is eligible to be discarded (returned to system pool).
CACHEFLUSH=*nnnn*	The CACHEFLUSH command sets the number of milliseconds that a cache block can remain dirty (modified) before the file system flushes it to the storage media.
CHAIN=*filename*	The CHAIN command transfers control to another configuration file. Because CONFIG.SYS is read in several passes, the CHAIN statement takes effect after all of the statements in CONFIG.SYS have been processed. Subsequent CHAIN commands override earlier ones. A common use of this command is to optionally execute several additional commands as an optional group by using the "?" character in front of the CHAIN command.
COMMENT=*any text*	The COMMENT command is used to add structured comments to the CONFIG.SYS file.
COUNTRY=*nnn*	The COUNTRY command sets the country code so that the DOSCOUNTRYINFO service can return this value to application programs.
DEVICE=*devname*	The DEVICE command loads a specified file as a device driver (either character or block) in the system. It is expected to conform to the architectural constraints set forth in the EMBEDDED DOS 6-XL Technical Reference Manual.
DEVICEHIGH=*devname*	The DEVICEHIGH command loads a specified file as a device driver in upper memory (such as a UMB), where it does not consume memory in the lower 640KB area.
ECHO=*any text*	The ECHO command is used to display a message to the console during CONFIG.SYS processing.
FCBS=*nnnn*	The FCBS command remains for compatibility with MS-DOS CONFIG.SYS files; it has no effect on operation of the system.

	EMBEDDED DOS 6-XL dynamically allocates system structures in response to application requests.
FILES=*nnnn*	The FILES command remains for compatibility with MS-DOS CONFIG.SYS files; it has no effect on operation of the system. EMBEDDED DOS 6-XL dynamically allocates system structures in response to application requests.
INSTALL=*progname*	The INSTALL command loads a specified file as a program as though it were executed in AUTOEXEC.BAT. This allows programs such as TSRs to be run without requiring COMMAND.COM or an AUTOEXEC.BAT file. SMARTDRV.EXE is commonly loaded with this command.
INSTALLHIGH=*progname*	The INSTALLHIGH command loads a specified file as a program in upper memory (such as a UMB), where it does not consume memory in the lower 640KB area.
IRQPRIORITY=*n*	The IRQPRIORITY command rotates the first programmable interrupt controller's interrupt priority so that the specified IRQ level has the highest priority in the system. For example, if IRQPRIORITY=3 is specified, then interrupts arriving on IRQ3 will have highest priority, followed by 4, 5, 6, and 7. Then, the 8259 automatically follows 7 with the lower priorities under 3, starting with 0 (the scheduling interrupt), 1, and 2 (the second PIC). This command should be used to assign the highest priority to COM ports when using communications packages in a multithreaded environment, or EMBEDDED DOS 6-XL will potentially context switch inside an ISR.
LASTDRIVE=*d:*	The LASTDRIVE command specifies the last drive letter to be used by DOS for its own purposes. Drive letters after this one are typically managed by network redirectors such as Novell NetWare. The default last drive is E:.
REM=*any text*	The REM command is used to add structured comments to the CONFIG.SYS file.
STACKS=*nnnn*	The STACKS command sets the number of INT 21h stacks allocated by EMBEDDED DOS 6-XL during system initialization. By default, EMBEDDED DOS 6-XL allocates 3 stacks, each taking 1K bytes of memory. INT 21h stacks are used to achieve parallelism inside EMBEDDED DOS 6-XL INT 21h calls. An INT 21h stack is automatically allocated by EMBEDDED DOS 6-XL when a thread calls INT 21h; if multiple threads call INT 21h simultaneously, then that many INT 21h stacks should be declared for maximum reentrancy.

	If insufficient stacks remain in the system for an INT 21h service to be executed, then it is postponed until an INT 21h stack becomes available.
STACKSIZE=*nnnn*	The STACKSIZE command sets the size in bytes of the stack to be allocated when an AllocateThread call is made by the application, or when an AllocateThreadLong call is made that defaults the stack segment. The stack pointer of a newly-allocated thread is set to this value. This parameter has a default value that is OEM-configurable in OEM.INC.
SHELL=*progname*	The SHELL command specifies the initial program to load when all system initialization has completed. By default, the shell program name is d:\COMMAND.COM. If you wish to have your own program loaded in place of COMMAND.COM, use the SHELL command and specify the program name, including the full path and extension (COM or EXE).
SYSTEMPOOL=*nnnnn*	The SYSTEMPOOL command sets the size of the system pool in bytes, up to a maximum value of 60000. By increasing this value, more kernel objects may be allocated in the system simultaneously. You may need to increase this value if you are using multiple threads, timers, open files, etc. in your embedded system. This parameter has a default value that is OEM-configurable in OEM.INC.
UMB=*xxxx*	The UMB command specifies the hexadecimal segment address of an upper memory block (UMB) made available by a memory manager. Using the UMB command registers the memory block with the EMBEDDED DOS 6-XL arena manager, enabling programs to acquire the memory through standard DOS services, and C Library services that call the DOS services; i.e., malloc.
VERIFY=ON|OFF	The VERIFY command sets the initial state of the DOS VERIFY flag, used by the FAT file system cache to enable the write-behind algorithm (delayed writes) if VERIFY=OFF. If VERIFY is set to ON, then the file system operates in a hardened mode (at the expense of reduced performance), making the file system more resiliant to application crashes, since dirty file system cache blocks are flushed to disk before each write completes. If VERIFY is set to OFF, then the automatic cache flushing on each I/O is disabled, causing the write-behind algorithm to write the data according to the CACHEFLUSH and CACHETTL parameter values.
VERSION=*n*	The VERSION command sets the version number that DOS reports to applications.

Device Driver Reference

EMBEDDED DOS 6-XL ships with installable device drivers that can be loaded with the DEVICE= and/or DEVICEHIGH= statements in CONFIG.SYS. This section documents their use.

ANSI.SYS Driver

Function: Emulates ANSI Escape Sequence Video Terminal.

Type: Character device driver

Syntax: DEVICE=ANSI.SYS

Parameters:

none.

Description:

The ANSI.SYS driver is loaded as a substitute for the default CON driver that is built into the EMBEDDED DOS 6-XL system. ANSI.SYS watches output requests and intercepts ANSI escape sequences emitted by programs that require an ANSI terminal in order to clear the screen, position the cursor, and perform other screen-oriented tasks. When these escape sequences are detected, they are translated by ANSI.SYS to the appropriate cursor positioning video BIOS requests.

DEBUG.SYS Driver

Function: Kernel Debugger.

Type: Character device driver

Syntax: DEVICE=DEBUG.SYS [/PORT=*n*]

Parameters:

n Specifies the COM port number (1, 2, 3, or 4) that the debugger should use to redirect its keyboard and screen to. Normally, when the port number is not specified, the debugger interacts with the PC's keyboard and screen through BIOS INT 10h and INT 16h requests.

Description:

The kernel debugger is used to provide the system integrator with the ability to debug system software running in the system. See the chapter on debugging with DEBUG.SYS for a thorough review of the debugger.

Example:

The following command instructs the debugger to use the keyboard and screen for debugging:

```
DEVICE=DEBUG.SYS
```

The following command instructs the debugger to use the COM1 port for debugging:

```
DEVICE=DEBUG.SYS  /PORT=1
```

IFSFUNC.SYS Driver

Function: MS-DOS Compatibility Driver.

Type: Character device driver

Syntax: DEVICE=IFSFUNC.SYS

Parameters:

none.

Description:

The IFSFUNC.SYS driver is used to more closely emulate MS-DOS internal undocumented structures so that programs that manipulate those structures will see increased undocumented functionality in EMBEDDED DOS 6-XL.

Specifically, this driver exposes the undocumented DOS CDS structure so that the application can edit the current directory of a drive and have that affect the operation of EMBEDDED DOS 6-XL.

In the future this driver may be used to provide increased compatibility with undocumented MS-DOS internals.

Example:

The following command loads IFSFUNC.SYS:

```
DEVICE=IFSFUNC.SYS
```

RAMDISK.SYS Driver

Function: Low Memory (Real-Mode) RAM Disk.

Type: Block device driver

Syntax: DEVICE=RAMDISK.SYS [/KBTOUSE=*nnn*]

Parameters:

nnn Specifies the number of kilobytes of main (low) memory to allocate for the RAM disk. This memory is taken away from the lower 640KB memory used for applications.

Description:

The RAMDISK.SYS driver provides emulation of a floppy disk drive of arbitrary size in low memory. EMBEDDED DOS 6-XL assigns the next available drive letter to the RAM disk.

Example:

The following command installs a 128KB RAM drive in the system:

```
DEVICE=RAMDISK.SYS  /KBTOUSE=128
```

SERDRIVE.SYS Driver

Function: Remote Disk Drive (via Serial Port).

Type: Block device driver

Syntax: DEVICE=SERDRIVE.SYS [/PORT=COMn] [/BAUD=baud

Parameters:

/PORT=COMn
Specifies the serial port being used for the remote disk drive. The default is COM1.

/BAUD=baud
Specifies the baud rate to use. Faster baud rates improve performance, but too fast a baud rate may cause disk errors. The valid values for baud are 115kb, 57.6kb, 28.8kb, 19.2kb, and 9600. If no baud rate is specified, the driver attempts to detect the baud rate of the host.

Description:

The SERDRIVE.SYS driver attempts to communicate with a host over the specified serial port, expecting to interact with INTERSVR.EXE, running on a host machine.

Once connected, SERDRIVE.SYS creates a new drive letter and redirects all I/O for that drive to the host via the serial communications line.

SERDRIVE.SYS automatically determines the host computer's communication speed, attempting 115Kbaud first, then 57600 baud, then 28800, 19.2K, and finally 9600. If the host does not respond at any of these baud rates, then I/O is not redirected and the device driver performs no function.

The baud rate for the communications line is established on the host side when running INTERSVR.EXE (see the command reference for details).

Example:

The following command causes SERDRIVE.SYS to redirect disk I/O to COM2 at 57.6K baud:

```
DEVICE=SERDRIVE.SYS  /PORT=COM2 /BAUD=57.6kb
```

The following command optionally loads SERDRIVE.SYS by using the '?' feature in CONFIG.SYS.

```
? DEVICE=SERDRIVE.SYS
```

Chapter 4

USER COMMAND REFERENCE

This chapter describes the user-level commands supported by the standard EMBEDDED DOS 6-XL command interpreter and utilities.

Basic User Concepts

Throughout the text of this guide, we will refer to various terms that have a specific meaning beyond their normal English meaning. You should be familiar with these terms and their associated concepts before using EMBEDDED DOS 6-XL commands.

Files, Directories, and Filenames

EMBEDDED DOS 6-XL allows you to create, delete, and manipulate objects called *files*. Files contain zero, one, or more bytes of information, and are stored electronically on floppy diskettes, hard disks, laserdisk, or other storage media. Every file has a name, or *filename*, and these names are stored in special system files called *directories*.

Files may be created and duplicated with the COPY command. The DEL and ERASE commands are used to delete files.

Directories can be created, deleted, and manipulated with the EMBEDDED DOS 6-XL command processor. The DIR command can be used to list the contents of a directory. The MKDIR command makes a new directory, and the RMDIR removes a directory.

Directories also have names, and these names are stored along with other files' names. The master directory is named "\", pronounced *root*. The root directory contains files and subdirectories, as do all of those subdirectories.

Filenames (and directory names) can be typed in uppercase or lowercase letters, and can also include special symbols such as '&' and '+', as well as numbers. For DOS-compatible file systems, filenames follow a rigid format that is a carry-over from the days of CP/M and MS-DOS:

```
filename.ext
```

where *filename* is a 1- to 8-character name, and *ext* is a 0- to 3-character extension to the name. Both parts are separated by a period (.) when the extension is specified. When no extension is specified, the period is not necessary, but may be specified.

Wildcarded Filenames

Normally, filenames (and directory names) refer to exactly one file; for example, when creating a directory with MKDIR and deleting files with DEL.

Some commands, such as COPY, DIR, and DEL, allow the use of a special form of a filename that lets you specify that some parts of the filename are "don't cares". These don't care symbols ('?' and '*') are called wildcards.

The '?' wildcard character can be used in place of any character in the name or extension parts of a filename, and tells EMBEDDED DOS 6-XL that you mean all files that match the filename without regard to that character. For example:

```
BILL.DOC
OPUS.DOC
BILLCAT.TXT
FDISK.COM
BASIC.EXE
BANANA
BILLFOLD.
```

are all valid filenames, and all of them but OPUS.DOC and FDISK.COM start with the letter 'B'. By refering to the wildcarded filename, "B???????.???", you can indicate all files that start with a letter 'B'. If you wanted all files that have an extension of "DOC", then you could use:

```
????????.DOC
```

Similarly, you could refer to all files (namely, BANANA and BILLFOLD.) that have no extensions by specifying the following wildcarded filename that has a period but no extension:

```
????????.
```

Because it is common to want to fill-out the remainder of either the name or extension parts of a wildcarded filename with the '?' wildcard character, EMBEDDED DOS 6-XL provides the '*' wildcard character to serve as a shorthand wildcard. This character instructs EMBEDDED DOS 6-XL to wildcard the remainder of the name or the extension (depending on which field contains the character). Thus, the following wildcarded filename specifies all files that begin with the letter 'B':

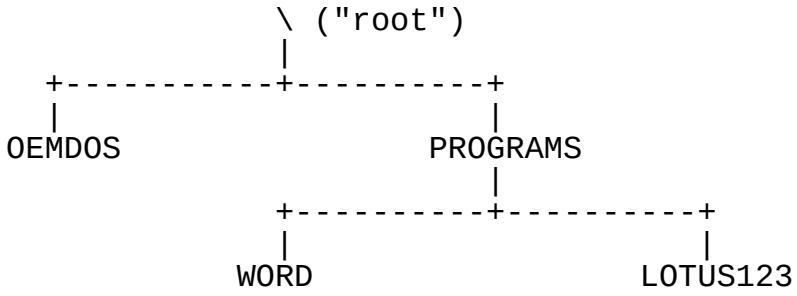
```
B*.*
```

Similarly, you could refer to all files that have a "DOC" extension with the following notation:

```
*.DOC
```

Pathnames

EMBEDDED DOS 6-XL stores filenames in directories, and those directories can contain subdirectories. Even so, the directory that contains all of the other files and subdirectories on a disk is the root directory, "\". This allows you to organize your files by using subdirectories that form a *tree* of directories and files. For example, if you have a word processor program, and a spreadsheet program, and your EMBEDDED DOS 6-XL program utilities, you might want to create a directory structure like the one below to organize your files:



This allows you to separate files into logical groups, and also allows you to create and maintain filenames that may by coincidence have the same exact filenames by keeping them in their own subdirectories. For example, your word processor and spreadsheet programs might both contain "READ-ME" files that could actually have the same filename. This subdirectory organization keeps them separate.

In our example, the PROGRAMS subdirectory contains two subdirectories, WORD and LOTUS123. In order to properly refer to the 123.EXE program in the LOTUS123 directory, you would need to specify a full *pathname* of that file that tells EMBEDDED DOS 6-XL where to find the file. The proper pathname for a file called 123.EXE located in the LOTUS123 subdirectory in our example is:

```
\PROGRAMS\LOTUS123\123.EXE
```

The Current Directory

Of course, with many nested subdirectories, specifying the full pathname of a file everytime it is used could become tedious. To help make the job of specifying files easier, EMBEDDED DOS 6-XL maintains a *current directory* for each drive in your workstation.

The current directory for any drive can be displayed with the following command:

```
C> CD d:
d:\PROGRAMS
C>
```

where d: is the name of the drive to display the current directory for. To change the current directory, simply specify the pathname of the directory to change to. For example, to change into the LOTUS123 directory of our example as shown above (assuming drive C), you would type the following:

```
C> CD C:\PROGRAMS\LOTUS123
C>
```

Thereafter, you could refer to the 123.EXE file in that directory simply as 123.EXE, instead of specifying the full pathname.

EMBEDDED DOS 6-XL knows that you are refering to a filename in the current directory when you leave off the backslash at the beginning of a filename. If you start a pathname with a backslash, then EMBEDDED DOS 6-XL assumes that you are specifying a pathname that starts from the root directory.

The Current Drive

Because most users usually use files on one drive (a hard disk, for example), EMBEDDED DOS 6-XL maintains a *current drive* for you so that, if you do not specify a drive letter in a pathname, EMBEDDED DOS 6-XL assumes that you mean the default drive.

When EMBEDDED DOS 6-XL starts, it sets the default drive to the one that it boots from. For example, if you booted your workstation with a floppy diskette in drive A, then the current drive would also be A by default. To change the default drive to another drive letter, simply type that drive letter followed by a colon (:) at the prompt. For example, if you booted from drive A but wanted to change to drive C afterward, you would type the following:

```
A> C:
C>
```

Putting it All Together

You can use all of the concepts in this section in combination when you use EMBEDDED DOS 6-XL. When constructing a filename, you must think about which components you will need to specify so that EMBEDDED DOS 6-XL fully understands where the file is located. The full form of a file specification is shown below:

```
[d:][pathname][filename][.[ext]]
```

The *d:* is an optional drive letter, and need only be specified if the file is located on a drive that is not the default drive.

The *pathname* field may omitted, in which case EMBEDDED DOS 6-XL assumes that the file is in the default directory. If you specify a pathname, and it starts with a backslash (\), then EMBEDDED DOS 6-XL knows that you are specifying a pathname that starts from the root directory. Otherwise, EMBEDDED DOS 6-XL treats the pathname as relative to the current directory.

The *filename* field is required; however, it may contain wildcard characters when the command supports it.

The *ext* field is optional, and if it is omitted, only those files with no extensions will match the filename. When omitted, the period may be specified or omitted. This field, like the *filename* field, can be wildcarded with '?' and/or '*'.

Special Filenames

EMBEDDED DOS 6-XL reserves some filenames for itself. In all systems, the following filenames are actually names of *devices* in the system, and cannot be used as filenames or pathnames to name your own files:

```
CON  - Console keyboard and screen
AUX  - First serial port
PRN  - First printer port
NUL  - Null device
COM1 - First serial port
COM2 - Second serial port
COM3 - Third serial port
COM4 - Fourth serial port
LPT1 - First parallel port
LPT2 - Second parallel port
LPT3 - Third parallel port
CLOCK$ - The system clock device
```

These names may be used when refering to these devices; however, you cannot name your files by these names. For example, if you wanted to copy data from a file called README.TXT to the printer connected to the first parallel port, you could use the following EMBEDDED DOS 6-XL command:

```
C> COPY README.TXT LPT1
```

Similarly, if you wanted to type some data from the console's keyboard into a file called MYFILE.DAT, you could use the following sequence:

```
C> COPY CON MYFILE.DAT
The men of the town go to the little island
to find coal for their city.  They go down
to the dock at eight and fight for a cozy
spot on the prow of the boat.
^Z
C>
```

Afterwards, you could use the following command to display the contents of the file:

```
C> TYPE MYFILE.DAT
The men of the town go to the little island
to find coal for their city.  They go down
to the dock at eight and fight for a cozy
spot on the prow of the boat.
C>
```

Alternatively, you could use the special device name CON with the COPY command to achieve the same effect:

```
C> COPY MYFILE.DAT CON
The men of the town go to the little island
to find coal for their city.  They go down
to the dock at eight and fight for a cozy
spot on the prow of the boat.
C>
```

EMBEDDED DOS 6-XL reserves some special extensions for its own use. Files having a "SYS" extension are a part of the operating system and cannot be changed. Some other extensions are reserved to indicate what types of data are in the files, and have special meaning to EMBEDDED DOS 6-XL:

Extension	Type of File
.BAT	Batch file, contains commands to run.
.EXE	Executable program, BINARY format.
.COM	CP/M compatible program, BINARY format.
.SYS	EMBEDDED DOS 6-XL system file or driver.
.DBG	EMBEDDED DOS 6-XL debugger symbol file.

Starting an EMBEDDED DOS 6-XL Session

To begin using EMBEDDED DOS 6-XL, simply boot your computer system from your EMBEDDED DOS 6-XL **SYSTEM** diskette, or from a fixed disk if you have installed the system on your hard disk. EMBEDDED DOS 6-XL will load and display a sign-on banner similar to the following (note that OEM adaptations of EMBEDDED DOS 6-XL may display slightly different sign-on banners):

```
General Software EMBEDDED DOS 6-XL [rev. 45]
Copyright (C) 1991-1996 General Software.  All rights reserved.

Current date is Mon  03-05-96
Enter new date (mm-dd-yy):_

Current time is 15:24:02.84
Enter new time:_

C>_
```

The system is now ready to accept commands. The "C>" is printed by the EMBEDDED DOS 6-XL command processor program, COMMAND.COM, and is called "*the prompt*." You can change the style of the prompt to include other information besides the current drive letter. To learn how to do this, consult the PROMPT command reference later in this chapter.

When the EMBEDDED DOS 6-XL command processor is loaded, it automatically scans your boot drive for the file, AUTOEXEC.BAT. If found, it reads commands from that file first, and when all of those commands are finished executing, it will display the prompt and read commands from the keyboard. If there is no AUTOEXEC.BAT file, then the command processor just asks for the date and time, and then prints the prompt right away.

Ending an EMBEDDED DOS 6-XL Session

To end your session with EMBEDDED DOS 6-XL, you must first exit the program that you are running **before** turning off your machine. This causes all files used by the currently running program to be closed properly.

When you exit the program, you should see the command prompt displayed by the EMBEDDED DOS 6-XL command processor:

```
C>_
```

The EMBEDDED DOS 6-XL Command Shell

The program that EMBEDDED DOS 6-XL runs automatically after booting is COMMAND.COM, the EMBEDDED DOS 6-XL command shell. You may substitute another program for the command shell by specifying the SHELL= statement in the CONFIG.SYS file.

The command shell, or command processor, is responsible for reading commands from the keyboard or from a file called a batch file and executing the commands by recognizing the verb of each command as a specific directive to do something. Most of this chapter is an alphabetical list of command names and a description of each command's functions.

The command processor has many features that help you use the resources of your workstation more efficiently. While most commands read their input keyboard and display output on the screen, the input and output functions can be redirected to files and even to other programs. COMMAND.COM also provides an *environment space* that allows you to configure the operation of the command shell to meet your specific needs. Finally, the command shell can read its input from a file, either through running COMMAND.COM as a program and redirecting its input and/or output, or by executing a *batch file*. These facilities are the subject of the next few sections.

Redirection

When a program or internal command is executed, the command shell routes the input of the executed program to the keyboard, and its output to the console screen. You can redirect either or both the standard input and output streams to files, so that a program reads its input from a file, and/or writes its output to a file.

To redirect a program's output file, simply follow the program's name and arguments with a '>' symbol, and the pathname to which output should be written. For example, if we wanted to save a directory listing to a file called SAVEDIR.TXT, we could do this:

```
C> DIR > SAVEDIR.TXT
C>
```

Similarly, we could use the SORT command to collect lines of text from the keyboard, and write the sorted results to a file called SORTED.TXT:

```
C> SORT > SORTED.TXT
This is the first line.
The second line.
The last line.
^Z
One file copied.
C>
```

To redirect the input of a program (such as SORT for example), just follow the command with a '<' symbol, and follow it with the name of the file from which input should be read. Suppose we wanted to sort our directory we created by redirecting DIR's output into SAVEDIR.TXT. Let's redirect SORT's input to that file and let the output go to the screen:

```
C> SORT < SAVEDIR.TXT


        8 File(s)   5230592 bytes free
 Directory of  C:\SRC\EMBEDDED DOS 6-XL\DOC
 Volume in drive C has no label
.            <DIR>      1-23-91   4:08p
..           <DIR>      1-23-91   4:08p
ARCH     DOC   483840  11-07-90   1:09p
NORMAL   BAK     1024   2-03-91   4:34p
NORMAL   STY     1024   2-03-91   4:35p
SAVEDIR  TXT      123   2-03-91  10:52p
USER     BAK   160256   2-03-91  10:35p
USER     DOC   162816   2-03-91  10:52p
```

Finally, both the input and output streams of a program may be redirected. Suppose we wanted to sort the same file, but write the sorted output to a file called SORTDIR.TXT, for later editing with a text editor:

```
C> SORT < SAVEDIR.TXT > SORTDIR.TXT
C>
```

Redirection is a powerful tool. Together with the pipe construct, redirection can be used as "glue" to bind modular programs together to create new, custom programs.

Piping

The following command sequence lists the contents of a directory into a temporary file, and then sorts the file, copying its output to the console screen:

```
C> DIR > FILE.TMP
C> SORT < FILE.TMP
```

The same effect can be achieved by "gluing" both programs together with a *pipe*. Using a pipe, the command processor automatically builds the temporary file and executes both commands for you. Here is the same sequence using piping:

```
C> DIR | SORT
```

Any number of programs may be piped together to form a pipeline. The command processor executes the commands sequentially; no multitasking actually occurs. Temporary files are created in the current directory of the default drive, and are removed when they are no longer needed.

The Environment Space

The command processor maintains an area of memory that contains variables and associated values. You can examine and change these variables with the SET command, detailed later in this chapter.

Application programs can also access this area, and commonly require setup information to be stored in the environment prior to being run. See your application program's owner's manual for a list of any environment variables that are supported by the application.

There are two very important variables used by the command shell itself; namely, COMSPEC and PATH.

The COMSPEC variable is assigned the full pathname, including boot drive, of the copy of the command processor that was loaded. This allows COMMAND.COM to reload itself under certain circumstances. Other programs may examine this variable to determine the drive on which utility programs might be located, for example.

The PATH variable is used to store the program search path, described next.

Program Search Path

When you run an application program, the command processor attempts to run the program from the current directory on the default drive. If it cannot be found in that directory, then it searches the PATH variable for other directories to try. The PATH variable is normally set to the following general format:

```
PATH=dir1;dir2;dir3;...;dirn
```

where *dir1* is the first directory to scan for the program, *dir2* is the second, and so on, until the last directory scanned is *dirn*.

In order for the command processor to always find its external utility programs, you may want to copy them to a directory called C:\GS. Then, you could set up the PATH variable to cause COMMAND.COM to scan that directory whenever it can't find a program name in the current directory:

```
C> SET PATH=C:\GS
```

Batch Files

EMBEDDED DOS 6-XL lets you store one or more commands in files with a special extension, BAT. These files are called *batch files*. By entering the filename of the batch file as a command, the command processor can run the commands in the file for you automatically.

Batch files can be very simple or extremely complex. A very simple batch file might delete all of the temporary files in the current directory that have a TMP extension:

```
C> COPY CON DELTEMP.BAT
del *.tmp
^Z
One file copied.
C> DELTEMP                (the batch file is run)
C>
```

To provide more flexibility, you can pass parameters to batch files on the command line. These parameters are made available in the form of special variable names; namely, %1, %2, %3, %4, %5, %6, %7, %8, and %9.

For example, if you wanted to implement a batch file that would move a file across directories or drives (the RENAME command cannot do this, as it is not strictly a renaming process), you might code the following lines in a batch file called MOVE.BAT:

```
COPY %1 %2
DEL %1
```

Sometimes you need to process a variable number of arguments on a batch file's command line. This requires a way to loop through all of the arguments, and processing them until an empty one is found.

For example, if you wanted to implement a batch file that would produce a directory listing of every argument that you pass on the command line (a "super directory" batch file), you might code the following lines:

```
:BIGLOOP
IF "%1"=="" GOTO ALLDONE
DIR %1
SHIFT
GOTO BIGLOOP

:EXIT
```

Another way of doing this might be:

```
FOR %%I IN (%1 %2 %3 %4 %5 %6 %7 %8 %9) DO DIR %%I
```

However the process is done, batch files can be powerful tools that automate daily chores. Once properly debugged, batch files can eliminate the human errors involved in backing up data, and many other tedious chores.

Alphabetical List of Commands

The remainder of this chapter describes each EMBEDDED DOS 6-XL command in great detail. Commands are categorized into two classes: External and Internal. External commands are actually separate utility programs supplied with EMBEDDED DOS 6-XL. They are loaded and run by the command processor when you type their name. Internal commands are built into the command processor and do not run any programs.

: Command

Function: Define Batch File Label.

Type: Internal

Syntax: :*label*

Parameters:

label Batch file label name

Description:

The : command marks a point of execution in a batch file with a name, or *label*, that can be used by the GOTO command to transfer control to the point in the batch file.

Looping and conditional execution with the IF command are possible through the use of labels defined with this command.

Lines containing a label definition may *not* contain another command itself; label definitions must occupy their own command line.

In interactive mode, label definitions are simply ignored.

Example:

The following command language fragment might be used in a batch file to loop repeatedly through all of the batch file's arguments.

```
:MAINLOOP
IF "%1"=="" GOTO EXIT
IF EXIST %1 ECHO The file %1 exists.
SHIFT
GOTO MAINLOOP

:EXIT
```

@ Command

Function: Disables Echo on Command Execution.

Type: Internal

Syntax: *@command*

Parameters:

command EMBEDDED DOS 6-XL command to be executed

Description:

The @ command executes the specified command in a batch file without echoing the prompt or the command itself, even when the ECHO flag is OFF.

The everyday use of this command is to disable the echoing of the ECHO OFF command at the start of a batch file. The @ command does not change the status of the ECHO flag itself, however.

Example:

The following command might be placed at the beginning of a batch file to disable all echo output within the batch file:

```
@ECHO OFF
```

ASK Command

Function: Prompts for and accepts operator input interactively.

Type: Internal

Syntax: ASK *varname=string*

Parameters:

varname Environment variable to set
string Prompt string

Description:

The ASK command displays the specified string on the console and waits for the operator to type a line of input at the prompt. When the operator presses the carriage return, ASK stores the typed-in text into the specified environment variable so that it may be inspected by an IF statement, or a program.

Example:

```
C> ASK COLOR=What color do you want?
What color do you want? RED
C> SET
COMSPEC=C:\COMMAND.COM
PATH=C:\DOS
COLOR=RED
C>
```

ATTRIB Command

Function: Displays, sets, or resets file attributes.

Type: External

Syntax: ATTRIB [+R | -R] [+A | -A] [*d:*][*pathname*]

Parameters:

+R	The file is marked read-only
-R	The file is marked read/write
+A	The file is marked for archiving
-A	The file is marked as unmodified
d:	Drive letter on which the specified files reside
pathname	Wildcarded pathname of files to affect or display attributes for

Description:

The ATTRIB utility program displays the attributes for a set of files, or can be used to change the read-only and archive attributes of the files.

If none of the +R, -R, +A, or -A parameters are specified, then the attributes of the specified files are displayed.

If the +R parameter is specified, then the files are marked read-only, so that they cannot be deleted inadvertently, and so that they are protected from accidental writing by application programs. Caution: Programs may modify the read-only attribute themselves, effectively reversing this protection.

If the -R parameter is specified, then any read-only marks are removed from the files' directory entries. This has the effect of allowing the files to be deleted with the DEL or ERASE commands, and the files can be overwritten by application programs.

If the +A parameter is specified, then the files are marked archivable, causing some backup programs to automatically select the program for backup.

If the -A parameter is specified, then the archivable marks are removed from the files, causing some backup programs to skip over the files when performing a full backup.

If the drive letter is specified, then ATTRIB searches that drive for the specified files. If no drive letter is specified, then ATTRIB uses the default drive.

ATTRIB requires you to specify an (optionally wildcarded) pathname that indicates which files are to be affected. Both the * and ? wildcard characters can be used.

Example:

```
C> ATTRIB +R *.WKS
C>
```

Batch Files

Function: Execute pre-recorded list of commands.

Type: External

Syntax: [*d:*][*pathname*] [*parameter1*] [*parameter2*] [...]

Parameters:

d: Drive letter on which the specified batch file resides
pathname Pathname of a file containing list of commands
parameter1, parameter2 - optional parameters

Description:

A batch file is a text file that contains zero, one, or more lines, each containing a valid EMBEDDED DOS 6-XL command as it would be typed manually at the system prompt. Once these commands are recorded in a batch file (with a text editor, for example), the entire sequence of commands may be executed by simply using to the batch file's name as a command name.

Batch files must always have a .BAT extension. When invoking the batch file, the extension is not used as a part of the command.

Up to ten parameters, may be passed to a batch file by simply including them as parameters to the command name. EMBEDDED DOS 6-XL automatically assigns them special numeric names that can be used in commands that are inside the batch file. When the commands in the batch file are executed by the EMBEDDED DOS 6-XL command processor, the special names are automatically expanded to the names you specify on the command line. The names of the parameters consist of a percent sign, followed by a number, starting from 1 for the first one. For example:

```
C> COPY CON COPYIT.BAT
copy %1 %2
^Z
C> COPYIT thisfile.dat theother.dat
One file copied.
C>
```

Environment variables may also be passed to batch files through a similar syntax. To refer to the *value* of an environment variable, simply enclose the *name* of the variable (in upper case) with percent signs (%). For example:

```
C> SET DESSERT=CAKE
C> ECHO We are eating %DESSERT% tonight.
We are eating CAKE tonight.
C>
```

To stop a currently-executing batch file, you can press either CTRL-C or CTL-BRK. EMBEDDED DOS 6-XL will display the message:

```
Terminate batch job (Y/N)?
```

If you type a Y, then the batch file will stop executing and the EMBEDDED DOS 6-XL command processor will issue a prompt for you to input the next command from the keyboard. If you type N, then the interrupted program will be aborted, but the batch file will continue with the next command.

BREAK Command

Function: Displays or changes the status of the BREAK flag.

Type: Internal

Syntax: BREAK [ON | OFF]

Parameters:

ON	Turns on ^C and CTL-BRK checking
OFF	Turns off ^C and CTL-BRK checking

Description:

The BREAK command changes or displays how EMBEDDED DOS 6-XL handles break-ins by the console user with ^C and CTL-BRK key sequences. If BREAK is ON, then EMBEDDED DOS 6-XL will break out of a running program or batch file when the ^C or CTL-BRK keys are pressed. If BREAK is OFF, then EMBEDDED DOS 6-XL will not break out, but will instead pass the keys pressed to the program.

Turning the BREAK flag ON makes it much easier to terminate a run-away batch file or program; however, some programs may not provide adequate safeguards against break-ins by the console user. Turning the flag OFF improves the protection of running batch files and programs, but makes it very difficult to terminate them if they run away.

Examples:

If no parameters are specified on the BREAK command, then the status of the BREAK flag is displayed. For example:

```
C> BREAK
```

```
BREAK is ON.
C>
```

If ON or OFF is specified, then the BREAK flag is set to that value. For example:

```
C> BREAK OFF
C> BREAK
BREAK is OFF.
C>
```

CALL Command

Function: Execute pre-recorded list of commands as a subroutine.

Type: Internal

Syntax: CALL [*d:*][*pathname*] [*parameter1*] [*parameter2*] [...]

Parameters:

d: Drive letter on which the specified batch file resides
pathname Pathname of a file containing list of commands
parameter1, parameter2 - optional parameters

Description:

The CALL command executes a batch file as described in "Batch Files". Unlike simply executing the batch file by invoking its name, the CALL command executes the specified batch file as a subroutine, so that one batch file can call another batch file.

```
C> COPY CON COPYIT.BAT
copy %1 %2
call DONE.BAT
^Z

C> COPY CON DONE.BAT
echo We're done.
^Z

C> COPYIT thisfile.dat theother.dat
One file copied.
We're done.
C>
```

CHDIR Command

Function: Displays or changes a drive's current directory.

Type: Internal

Syntax: CHDIR [*d:*][*path*]

Parameters:

d:	Drive letter to change/display the current directory for
path	New current directory for the specified drive

Description:

The CHDIR command (abbreviated CD) displays the current directory of the specified drive, or can change the current directory of the specified drive. If no drive is specified, then the default drive is used.

Each volume in the system is assigned a drive letter by EMBEDDED DOS 6-XL at system initialization time. Each drive letter has associated with it a "current" directory pathname, which is used internally by EMBEDDED DOS 6-XL to interpret pathnames that are not fully specified. When the system starts, the current directory of each drive is initialized to the root directory (\).

The current directory of a drive may be changed without changing to that drive itself by specifying a drive letter.

The current directory of the default drive may be changed by not specifying the drive letter.

If no parameters are specified, then the current directory is displayed for the default drive. If only a drive letter is specified, then the current directory is displayed for the specified drive.

Examples:

The following command displays the current directory of the default drive:

```
C> CHDIR
C:\
C>
```

The following command changes the current directory of the current drive to the WINDOWS subdirectory under the root directory:

```
C> CHDIR WINDOWS
C>
```

The following command changes the current directory of drive E to the root (top-level) directory:

```
C> CHDIR E:\
C>
```

CHKDSK Command

Function: Analyzes and maintains file system integrity.

Type: External

Syntax: CHKDSK [*d:*][*pathname*] [/F] [/V]

Parameters:

d:	Drive letter containing file system to inspect
pathname	Wildcarded pathname of files to analyze for contiguity
/F	Fixes, or corrects, any file system errors
/V	Displays each file as it is analyzed by CHKDSK

Description:

The CHKDSK utility program analyzes the file system residing on a drive to determine if it contains structural errors. At the user's option, CHKDSK can be instructed to correct the structural errors.

If no parameters are specified, then CHKDSK inspects the default drive, but does not check any files' contiguity, nor does it fix any structural problems. CHKDSK displays a list of structural errors as they are found, and performs no corrective action.

If a drive letter is specified, then CHKDSK will operate on the specified drive. Because EMBEDDED DOS 6-XL may assign drive letters to non-DOS file system types (including NetWare and OS/2 HPFS, for example), the file system must be supported by CHKDSK before it can correctly analyze the file system's structure. If the file system is not one of the supported types, CHKDSK will display a message to that effect, and will not check the file system.

If a pathname with optional wildcard characters (* and ?) is specified, then those files are checked by CHKDSK to determine if they are stored contiguously on the storage media. Files that are stored in one chunk can be processed by EMBEDDED DOS 6-XL more efficiently than those that are fragmented into many pieces. This option can be used to determine if the performance of a program such as a database manager can be improved simply by copying the file to a new location in the file system.

If the /F option is specified, then CHKDSK will correct any structural damage to the file system. Because this action is highly file system-specific, a list of actions is not given here. Generally, CHKDSK will determine that files do not accidentally share allocated storage blocks, and that their sizes as recorded in the directory are consistant with the allocated storage blocks for those files.

If the /V option is specified, then CHKDSK will display the name of each file as it is analyzed. This allows the user to observe the progress of CHKDSK as its operations commence.

Before terminating, CHKDSK reports general statistics about the file system's storage capacity, and the amount of space used by different types of file system objects such as directories, files, hotfix areas, and so on.

CLOSE Command

Function: Closes a file opened with the OPEN command.

Type: Internal

Syntax: CLOSE *file*

Parameters:

file Environment variable assigned to file at OPEN time

Description:

The CLOSE command closes a file that was previously opened with the OPEN command. Any data buffered by the command shell is flushed to the operating system, although the data may be lazy-written to the file system at the system's option.

After the file has been closed, the environment variable is removed from the environment.

If the environment variable does not represent a valid open file, then an error message is displayed and the environment variable is not altered.

Examples:

The following example opens a file called TANGO.DAT, reads a line of text from it into the variable called BOUNTY and then closes the file:

```
C> OPEN TANGO TANGO.DAT
C> READ TANGO BOUNTY
C> CLOSE TANGO
```

CLS Command

Function: Clears the screen.

Type: Internal

Syntax: CLS

Parameters:

none.

Description:

The CLS command clears the workstation's screen and resets the cursor position to the upper-left hand corner of the screen. The next prompt is issued on the top line of the screen.

COMMAND Command

Function: Runs the EMBEDDED DOS 6-XL command shell.

Type: External

Syntax: COMMAND [/E:*nnnn*] [/P] [/S] [/C *string*]

Parameters:

nnnnn Maximum environment size in bytes.
string Command with arguments to execute.

Description:

The COMMAND program utility is a command shell that actually runs as an application program. When EMBEDDED DOS 6-XL boots, it runs COMMAND as the default shell, unless the user places a SHELL= statement in the CONFIG.SYS file that directs EMBEDDED DOS 6-XL to another shell program.

COMMAND reads commands, one per typed line, from standard input, and writes its messages to standard output.

If the /E option is specified, then the user can change the environment size to be used while the command processor is executing. The environment should be greater than 128 bytes, but less than 32,768 bytes.

If the /S option is specified, then COMMAND will assume that it is operating over a serial port, and will not employ the use of the "task bar" as a part of the command line prompt.

If the /P option is specified, then COMMAND will not allow the EXIT statement to terminate the shell. Instead, the EXIT command does nothing when typed on the keyboard. This option is not intended for user application; instead, it is used by EMBEDDED DOS 6-XL to tell the command shell to remain in memory when EXIT is typed.

If the /C option is specified, then COMMAND loads into memory and executes the specified string as a command. The prompt is not displayed, and COMMAND terminates as soon as the specified command string is finished executing.

Example:

The following example runs the CHKDSK utility within another process and then returns.

```
C> COMMAND /C CHKDSK B:
```

COPY Command

Function: Copies one or more files or directories.

Type: Internal

Syntax: COPY [*d:*][*path1*] [*d:*][*path2*] [/V] [/A] [/B] [/R]

Parameters:

d:	Drive to copy files from/to.
path1	File(s) or directory to copy from.
path2	Target pathname(s) or directory.
/V	Disables the write-behind cache during the copy.
/A	Copies ASCII files up to end-of-file markers.
/B	Copies BINARY files without interpreting EOF markers.
/R	Copies recursively into subdirectories.

Description:

The COPY command copies one or more files (and optionally directories, with the /R option) to a new destination. If the destination path names a file, then all of the source files are written to the target file, concatenated together.

If *path2* is not specified, the copy is created in the current directory on the disk in the default drive. If the source file (*path1*) is also on the default drive, and *path2* is not specified, then an error message is displayed, indicating that a file cannot be copied to itself.

If the /V option is specified, then COPY temporarily turns VERIFY mode ON, so that the data will be written directly to the disk, and not into the cache, where it remains volatile until written by the system. By default, COPY leaves VERIFY in the state specified by the user, which defaults to OFF. This normally has the effect of improving COPY's performance by writing to the cache instead of the disk.

If the /A option is specified, then the files will be copied in cooked mode; that is, they are assumed to contain ASCII text, and will only be copied to the first EOF mark (an embedded control-Z character). This is the default when files *are* concatenated.

If the /B option is specified, then the files will be copied in raw mode; the entire files are copied without regard to their contents. This is the default when files are *not* concatenated.

If the /R option is specified, then COPY recursively descends into the source file list, copying all of the subdirectories and their contents to the target. This option works similarly to the XCOPY command except that it works without loading another program.

Examples:

The following command copies the EMBEDDED DOS 6-XL system file DOS.SYS from drive A to the root directory of drive C:

```
C> COPY A:DOS.SYS C:\DOS.SYS
One file copied.
C>
```

The following command copies the EMBEDDED DOS 6-XL COMMAND.COM shell from the current directory of drive A to the current directory of drive C:

```
C> COPY A:COMMAND.COM
One file copied.
C>
```

CTTY Command

Function: Changes the default console device.

Type: Internal

Syntax: CTTY *device*

Parameters:

device New console device name to accept commands from.

Description:

The CTTY command redirects all input and output to another device in the system; for example, AUX or COM1. This allows a remote user on a modem, for example, to control the machine as though he had direct access to the workstation's keyboard and screen.

Many programs do not use API calls to display information on the screen or accept characters from the keyboard; these programs will not operate correctly when run remotely using this method.

Examples:

The following command allows the user of a remote "dumb" terminal connected to the COM2 serial port to enter commands and run programs remotely over the serial port:

```
C> CTTY COM2
```

The following command is used by the remote user to transfer control back to the console user:

```
C> CTTY CON
```

DATE Command

Function: Displays or changes the date.

Type: Internal

Syntax: DATE [*mm-dd-yy*]

Parameters:

mm-dd-yy New date to set the workstation's clock to.

Description:

The DATE command displays the current date (month, day, date, and year) on the screen. If a user specifies a new date on the command line, then DATE will change the date to the one specified.

EMBEDDED DOS 6-XL checks the date to ensure that it is reasonably correct. For example, February 31 does not exist; therefore, EMBEDDED DOS 6-XL would reject 02-31-95 because February does not contain 31 days.

This command actually sets the real-time clock in the workstation, if one exists. In AT-class machines, it updates the battery-maintained clock so that the new date will be remembered across power-downs.

Example:

The following example displays the current date and prompts the user for a new date. The user can press the ENTER key to keep the date the way it is:

```
C> DATE
Current date is Fri  03-03-95
Enter new date (mm-dd-yy):_
```

DEL Command

Function: Deletes one or more files.

Type: Internal

Syntax: DEL [*d:*][*path*]

Parameters:

d:	Drive on which the files are to be deleted.
path	Wildcarded pathname of files to be deleted.
/R	Recursively deletes files in subdirectories

Description:

The DEL (synonym ERASE) command deletes one or more files from a file system on a specified drive. If the specified path is a directory, all files in that directory will be deleted.

If path contains wildcards, then all files that match the wildcarded specification will be deleted.

If the /R option is specified, then DEL will recursively descend into each subdirectory specified in the path and delete *all* files in the subdirectories. The subdirectories themselves are also deleted when this option is specified.

If DEL is asked to delete a whole directory of files, then it asks the user to verify that it is okay to do so with the prompt:

```
Are you sure? (Y/N):
```

Warning: You should be very careful when issuing this command with wildcard path specifications, unless you are very familiar with the rules for expanding wildcards. Many files can be accidentally erased with a misplaced '*' character, for example.

Examples:

The following example deletes all files in the current directory of drive E:

```
C> DEL E:*.*
Are you sure? (Y/N): Y
C>
```

The following example deletes the file named FINANCE.WKS in the current directory of the default drive:

```
C> DEL FINANCE.WKS
C>
```

The following example deletes all of the files ending in the extension, ".BAK" in the current directory:

```
C> DEL *.BAK
C>
```

DELAY Command

Function: Delays a batch file for a specific machine-independent interval of time.

Type: Internal

Syntax: DELAY *seconds*

Parameters:

seconds Number of seconds to delay

Description:

The DELAY command pauses a batch file for a machine-independent amount of time. The system's time of day clock is queried to determine the time so that it will work on machines with different CPU speeds.

Examples:

The following example delays for 10 seconds before returning to the next command:

```
C> DELAY 10
C>
```

DELTREE Command

Function: Deletes all of the files and subdirectories inside a specified subdirectory.

Type: External

Syntax: DELTREE [*d:*][*path*

Parameters:

d: Drive on which the files are to be deleted.
path Wildcarded pathname of directory containing files to be deleted.

Description:

The DELTREE command deletes an entire subtree of the file system on the specified drive. All of the files in the subdirectory, all of the subdirectories of the subdirectory, and all of the files and subdirectories in those subdirectories, are removed.

Note that this command can delete lots of information quickly. Please take care when using it to avoid deleting the wrong data inadvertently.

Examples:

The following example deletes all files in the MYDIR subdirectory on drive C:

```
C> DELTREE C:\MYDIR
C>
```

DIR Command

Function: Lists the files in a directory or subdirectory.

Type: Internal

Syntax: DIR [*d:*][*path*] [/P] [/W]

Parameters:

d:	Drive on which the files are to be displayed.
path	Wildcarded pathname of files to be displayed.
/P	Pauses the display after each screenful of information.
/W	Selects a wide, 5-up display of filenames.

Description:

The DIR command displays the files and subdirectories in a directory on the specified drive. If the drive is not specified, then the default drive is assumed.

DIR uses the path operand to determine which files to list. If the path is not specified, then the current directory is assumed. If the specified path is a directory name, then all files in that directory are listed. If the specified path is a wildcarded filename, then all files matching the path specification are listed.

The /P option can be used to pause the display after each screenful of directory listing. This can be useful for displaying the contents of very large directories with hundreds of files.

The /W option is used to generate a wide listing that omits the date, time, and size of each file's listing. This makes room for several files to be listed on each line, enabling 5 directory entries to be printed on each line.

Examples:

The following example displays a list of all of the files in the current directory of drive E:

```
C> DIR E:

 Volume in drive C has no label
 Directory of  C:\SRC\GS\DOC

.            <DIR>      1-23-91   4:08p
..           <DIR>      1-23-91   4:08p
ARCH     DOC   483840  11-07-90   1:09p
USER     DOC   162816   2-03-91  10:52p
NORMAL   STY     1024   2-03-91   4:35p
USER     BAK   160256   2-03-91  10:35p
NORMAL   BAK     1024   2-03-91   4:34p
        7 File(s)   5230592 bytes free
C>
```

The following example displays a list of files that have ".DOC" extensions in the current directory of the default drive:

```
C> DIR *.DOC

 Volume in drive C has no label
 Directory of  C:\SRC\GS\DOC

.            <DIR>      1-23-91   4:08p
..           <DIR>      1-23-91   4:08p
ARCH     DOC   483840  11-07-90   1:09p
USER     DOC   162816   2-03-91  10:52p
        2 File(s)   5230592 bytes free
C>
```

DISKCOMP Command

Function: Compares two diskettes, track-for-track.

Type: External

Syntax: DISKCOMP [*d:*] [*d:*] [/1] [/8]

Parameters:

d:	Specifies the drives to compare.
/1	Compares one side of the disks only.
/8	Compares only the first 8 sectors of each track.

Description:

The DISKCOMP utility program compares the contents of two diskettes, track-by-track, until all of the tracks have been compared. If the disks are identical, then DISKCOMP displays the following message:

```
Diskettes compare OK
```

Otherwise, DISKCOMP displays a compare-error message that indicates which tracks do not match.

If the diskettes are not the same size or if they do not have the same capacity, then the following message is displayed:

```
Diskette types incompatible
```

DISKCOMP will ask if more diskettes are to be compared after each compare, enabling it to compare a batch of diskettes without re-running the utility program.

DISKCOMP won't work on diskettes that contain odd-sized sectors or unusual sector address marks (such as those employed by copy-protection schemes). DISKCOMP only copies 512-byte sectors.

A return code is stored in ERRORLEVEL upon exit, based on the compare status. The following values are returned:

0	Successful comparison.
1	Compare error.
2	Operator abort.
3	Hard error (disk unformatted, etc.)
4	Invalid parameters, or insufficient memory available.

Example:

The following example compares the diskette in drive A with the diskette in drive B:

```
C> DISKCOMP A: B:
```

DISKCOPY Command

Function: Copies one floppy diskette to another diskette.

Type: External

Syntax: DISKCOPY [*d1:*] [*d2:*] [/1]

Parameters:

d1:	Specifies the drive to copy from.
d2:	Specifies the drive to copy to.
/1	Copies one side only.
/F	Forces reformatting of target.

Description:

The DISKCOPY utility program copies the contents of one diskette to another, track-by-track, until all of the tracks have been copied. The second diskette is formatted as the copy proceeds, if it is determined to be of an incorrect format, or if it is unformatted.

If the /1 option is selected, then only one side (head 0) of the source diskette will be copied to the destination diskette. If the second diskette requires formatting, then only one side of the diskette will be formatted.

If the /F option is selected, then the target diskette will be formatted, even if it is determined to be pre-formatted. This is useful when a target disk has lost part of its formatting or has been incorrectly or partially formatted.

DISKCOPY asks the user if more disks are to be copied, enabling it to be used in copying batches of diskettes from the same master without reloading the utility program for every diskette.

A return code is stored in ERRORLEVEL upon exit, based on the compare status. The following values are returned:

0	Successful copy.
1	Recoverable read/write error occurred.
2	Operator abort.
3	Hard error (disk unformatted, etc.)
4	Invalid parameters, or insufficient memory available.

Example:

The following example copies the diskette in drive A to the diskette in drive B:

```
C> DISKCOPY A: B:
```

ECHO Command

Function: Displays/changes ECHO mode, displays messages.

Type: Internal

Syntax: ECHO [ON | OFF | . | *message*]

Parameters:

ON	ECHO mode should be turned on.
OFF	ECHO mode should be turned off.
.	"ECHO." as a command prints a blank line.
message	Message to be displayed.

Description:

The ECHO command has two functions; namely, control of the ECHO flag, and displaying messages in batch files.

ECHO mode controls the command processor's echoing of commands in batch files. If ECHO mode is on, then commands read from batch files are automatically echoed to the screen before they are executed. If ECHO mode is off, then commands are not echoed as they are executed. To display the current ECHO flag status, use the ECHO command without any parameters.

To display a message from a batch file, use the ECHO command with a non-empty string to be displayed. The special form of the ECHO command with a period ('.') immediately following the word ECHO (no intervening space) causes a blank line to be echoed.

Examples:

The following example displays the current setting of the ECHO flag:

```
C> ECHO
ECHO is ON.
C>
```

The following example displays the user message "Starting file copies . . ." on the screen when executed in a batch file:

```
ECHO Starting file copies . . .
```

ERASE Command

Function: Deletes one or more files.

Type: Internal

Syntax: ERA[SE] [*d:*][*path*]

Parameters:

d:	Drive on which the files are to be deleted.
path	Wildcarded pathname of files to be deleted.
/R	Recursively deletes files in subdirectories

Description:

The ERASE (synonym DEL) command deletes one or more files from a file system on a specified drive. If the specified path is a directory, all files in that directory will be deleted.

If path contains wildcards, then all files that match the wildcarded specification will be deleted.

If the /R option is specified, then ERASE will recursively descend into each subdirectory specified in the path and delete *all* files in the subdirectories. The subdirectories themselves are also deleted when this option is specified.

If ERASE is asked to delete a whole directory of files, then it asks the user to verify that it is okay to do so with the prompt:

```
Are you sure? (Y/N):
```

Warning: You should be very careful when issuing this command with wildcard path specifications, unless you are very familiar with the rules for expanding wildcards. Many files can be accidentally erased with a misplaced '*' character, for example.

Examples:

The following example deletes all files in the current directory of drive E:

```
C> ERASE E:*.*
```

```
Are you sure? (Y/N): Y
C>
```

The following example deletes the file named FINANCE.WKS in the current directory of the default drive:

```
C> ERASE FINANCE.WKS
C>
```

The following example deletes all of the files ending in the extension, ".BAK" in the current directory:

```
C> ERASE *.BAK
C>
```

EXIT Command

Function: Exits the command shell utility program.

Type: Internal

Syntax: EXIT

Parameters:

none.

Description:

The EXIT command terminates the current command shell and reverts control to the previous shell, provided that the current command shell is not the first one loaded in the system. The very first shell cannot be terminated with EXIT.

If executed from a batch file, EXIT will terminate the batch file in a controlled manner, causing control to be transferred to the keyboard user.

FDISK Command

Function: Partitions a hard disk into one or more volumes.

Type: External

Syntax: FDISK [/R]

Parameters:

/R Read-only mode; partitions are *not* affected.

Description:

The FDISK utility program is used during the installation or reconfiguration of a hard disk to view, install, or modify the disk's partition structure. In read-only mode, FDISK is inhibited from changing disks, allowing the user to view the status of the disk partitions.

Unlike its MS-DOS counterpart, FDISK can create and manage many different partition types, even non-DOS ones. The following types are supported:

- _ 12-bit FAT (MS-DOS compatible)
- _ 16-bit FAT (MS-DOS compatible)
- _ Extended DOS (MS-DOS compatible)
- _ Huge FAT (MS-DOS 4, 5, and 6 compatible)
- _ OS/2 FSD (HPFS and others)
- _ OS/2 mirror partitions
- _ NetWare 286 (Novell compatible)
- _ NetWare 386 (Novell compatible)
- _ Xenix partitions (SCO-Xenix compatible)

FDISK is a very simple menu-driven program that operates in a similar manner to its MS-DOS cousin.

Caution: When using FDISK to create or delete partitions, you risk loosing vast quantities of storage on your disk(s) with a slip of your finger. Be very careful when preparing your fixed disks with FDISK.

Be aware that FDISK numbers your hard drives starting with the number 0, and then 1, and so on. Be careful that you properly select the correct disk before editing its partition table.

FIND Command

Function: Scans one or more files for a text string.

Type: External

Syntax: FIND "*string*" [*d:*][*path*] [/V] [/C] [/N]

Parameters:

d:	Drive on which the files to be searched reside.
path	Wildcarded pathname of files to be scanned.
/V	Displays every line that does not contain the string.
/C	Counts the number of lines that contain the string.
/N	Numbers lines that are displayed.

Description:

The FIND utility program is used to scan a set of files for a text string line-by-line. If no extra options (/V, /C, or /N) are specified, then FIND simply displays each line that matches, as they are found.

If the /V option is specified, then FIND searches for lines that do *not* contain the specified text string.

If the /C option is specified, then FIND counts the number of matches (or mismatches, if /V is specified) that occur, and the number is displayed instead of the matching lines.

If the /N option is specified, then FIND displays line numbers in front of each matching (or mismatching) line as they are printed.

FIND is a member of a family of utilities known as "filters", that accept volumes of data and transform the data somehow, producing modified, reduced, or resulting, data on output. Filters are frequently connected together in the form of a "pipeline", as discussed in the section on piping and redirection, in this user's guide.

Examples:

The following example displays all the lines in the file PHONE.TXT that contain the string, George Bush:

```
C> FIND "George Bush" PHONE.TXT
```

The following example counts the lines in the file HOTSELL.LST on drive F that contain the string "DOS":

```
C> FIND /C "DOS" F:HOTSELL.LST
```

FOR Command

Function: Repeatedly executes a command over a range of arguments.

Type: Internal

Syntax: FOR *var* IN (*set*) DO *command*

Parameters:

var	Name of a variable to use.
set	Set of values to assign to *var*.
command	Command to execute for each value in *set*.

Description:

The FOR command executes a command several times, each time assigning a value from a set into a variable name. The variable name may optionally be used in the command as desired.

The variable name must start with a percent sign (%) if typed directly from the keyboard, or by two percent signs if executed from a batch file. Variable names are restricted to a single alphabetic letter.

The set consists of zero, one, or more (optionally wildcarded) pathnames. The FOR command automatically expands any wildcarded pathnames as it executes.

Any occurrences of the variable in the command will be substituted with the current value of the variable before it is run. The command may be an internal, external, or batch file command.

Examples:

The following example types out all of the files in the current directory of the default drive:

```
C> FOR %I IN (*.*) DO TYPE %I
```

This example copies all of the files having a DOC prefix to a BACKUP directory:

```
C> FOR %I IN (*.DOC) DO COPY %I \BACKUP
```

FORMAT Command

Function: Formats a diskette or hard disk partition for file system use.

Type: External

Syntax: FORMAT [*d:*] [/1] [/4] [/8] [/T:*tracks*] [N:*sectors*] [/V] [/S]

Parameters:

d:	Drive or partition to format.
/1	Diskette is to be formatted single-sided.
/4	Diskette is to be formatted double-density.
/8	Diskette is to be formatted 8 sectors per track.
/T:*tracks*	Number of tracks to be formatted.
/N:*sectors*	Number of sectors per track to format.
/V	Installs a volume label on the new file system.
/S	Copies EMBEDDED DOS 6-XL system files.

Description:

The FORMAT utility program prepares a hard disk partition, or a floppy diskette, for use with EMBEDDED DOS 6-XL. While hard disks are typically low-level formatted at the factory, floppy diskettes are usually shipped completely degaussed and the track structure must be established on the media. FORMAT initializes the track structure on floppy diskette media.

FORMAT also installs the skeleton of a file system (albeit an empty one) on the partition or floppy diskette so that it can be used to store files. The directory structure is initialized, and the disk blocks are all freed and readied to accept file data.

If the /S option is specified, then FORMAT will also copy the EMBEDDED DOS 6-XL system files (DOS.SYS, COMMAND.COM, and the partition boot record (PBR)) to the target partition or floppy diskette. This allows the partition or floppy diskette to become bootable by itself.

If the /V option is specified, then FORMAT also prompts the user for an 11-character volume name, which it installs in the file system as an electronic marker that labels file system in the same way that the LABEL command does.

If the drive is not specified, then the default drive will be formatted. Otherwise, the specified drive will be formatted. Caution: EMBEDDED DOS 6-XL assigns drive letters to all hard disk partitions, even non-DOS ones. This permits the system to support these file systems so that DOS applications can make use of them directly. You should be very careful to format the correct partition.

If /1 is specified, then FORMAT will only format one side of the floppy diskette. This option is not supported for hard disk partitions.

If /8 is specified, then FORMAT assumes that it should format each track with eight sectors per track. This option is not supported for hard disk partitions.

If /T:*tracks* is specified, then FORMAT only formats the specified number of tracks, enabling a portion of a diskette to be formatted.

If /N:*sectors* is specified, then FORMAT will organize each track into the specified number of sectors each. This option is designed to support new diskette media whenever it becomes available. Not all values are supported for all media types. This option is not supported for hard disk partitions.

Examples:

The following example formats a double-sided, double-density floppy disk with a standard, MS-DOS compatible, 360Kb capacity file system:

```
C> FORMAT A: /4
```

The following example formats drive D and installs the system on the drive:

```
C> FORMAT D: /S
```

GOTO Command

Function: Jumps to a label in a batch file.

Type: Internal

Syntax: GOTO *label*

Parameters:

label Name of a label to jump to.

Description:

The GOTO command causes the command processor to start executing commands that follow the specified label, in the current batch file. Labels can be inserted anywhere in batch files, and take the following form:

```
:label
```

Example:

For example, to create an infinite loop in a batch file that continuously scans for the existence of a file in a directory on a network drive N, you could use the following batch commands:

```
:MYLOOP
IF NOT EXIST TESTFILE.DAT GOTO MYLOOP
ECHO The file exists!  We are all done.
```

HELP Command

Function: Displays a list of available commands.

Type: Internal

Syntax: HELP

Parameters:

none.

Description:

The HELP command displays a list of the commands that are supported by the command interpreter. This is useful to a user that must use several different versions of DOS that support extensions provided by OEMs.

Examples:

The following example displays a list of commands supported by COMMAND.COM:

```
C> HELP
Available command list:

break   cd      chdir   cls     copy    date    del     dir
era     erase   exit    md      mkdir   path    prompt  ren
rename  rd      rmdir   set     time    type    ver
verify  vol     help    rem     ask     echo    goto
```

```
switch  for     if      pause   shift   call    delay   open
close   read    write   rewind  synch

C>
```

IF Command

Function: Executes a command based on a condition.

Type: Internal

Syntax: IF [NOT] *condition command*

Parameters:

condition	Condition that must be satisfied.
command	Command to be executed if condition is TRUE.

Description:

The IF command causes a command to be executed if (or if NOT) a condition is TRUE.

The condition can take any of three forms:

_ Testing the value of ERRORLEVEL:

```
ERRORLEVEL level
```

This condition is TRUE when the last external command returned the specified status. See DISKCOPY and DISKCOMP for examples of two commands that do this.

_ Testing the existence of a file:

```
EXIST [d:][path]filename[.ext]
```

This condition is TRUE only if the specified file exists.

_ Testing equivalence of two strings:

```
string1 == string2
```

This condition is TRUE when *string1* is lexically the same as *string2*. If multiple words (separated by spaces, tabs, or other separators) are needed, then you should place the strings in quotes, as follows:

"*string1*" == "*string2*"

Examples:

The following command prints "The file exists" if the file TEST.DAT exists:

```
C> IF EXIST TEST.DAT ECHO The file exists
The file exists
C>
```

The following command tests if the previous DISKCOPY command was successful or not:

```
C> IF ERRORLEVEL 0 ECHO The copy was successful
The copy was successful
C>
```

This command compares the first batch file argument in a batch file with a switch "/P" and if specified, transfers to the EXIT label in a batch file:

```
IF "%1"=="/P" GOTO EXIT
```

INTERSVR Command

Function: Enables disk, and optionally console, redirection through a serial communications line to a target computer. This allows the target computer access to a selected drive of the host computer, and optionally routes all keyboard and screen activity from the target computer's keyboard and screen to the host computer's keyboard and screen.

Type: External

Syntax: INTERSVR [/X=*d*:] [/BAUD=*baud*] [/COM=*port*] [/WRITE]

Parameters:

d:	Drive to allow target to access.
baud	115200, 57600, 38400, 19200, or 9600.
port	1 for COM1, or 2 for COM2.
/WRITE	Specifies write access to drive is permitted.

Description:

The INTERSVR utility enables full-duplex, multiplexed disk and console I/O requests captured by the SERDRIVE.SYS device driver on a target computer to be serviced on a host computer. There are two benefits of this linkage:

1. The target has access to the host's hard drive.
2. The host can be used as the target's console.

Once INTERSVR.EXE is loaded, it waits for requests to be routed to it by the target computer. At any time, the host user may press the CTL and ALT keys down together for a couple seconds to terminate the communications link.

INTERSVR.EXE provides read-only access to the host's drive by default. If write access is desired, add the /WRITE option to INTERSVR's command line.

By default, communications are established at 57K baud, which is half the possible speed of the link. With proper cabling and matched UARTs on the host and target, it is possible to use the link at 115,200 baud, but this does not work under all circumstances. If you find that your cabling or host/target combination does not support communications at 57K baud, you may wish to try communications at 9600 baud first, and work up until the highest baud rate is determined.

While the host sets the baud rate, the target's SERDRIVE.SYS tries all of the baud rates at the time it connects to the host, from highest to lowest. If you switch baud rates on the host, it will be necessary to reboot the target so that it can find the new baud rate you have selected.

Example:

The following example establishes a link with a target on COM1, shares drive C:, and makes it writable:

```
C> INTERSVR  /X=C: /PORT=1  /WRITE
```

LABEL Command

Function: Creates or changes a volume label on a diskette or hard disk partition.

Type: External

Syntax: LABEL [*d:*] [*label*]

Parameters:

d:	Drive or partition to assign a label to.
label	New volume label to assign to the drive or partition.

Description:

The LABEL utility program deletes any existing label associated with the specified drive and optionally installs a new label.

Volume labels can be from one to 11 characters in length, and cannot contain any of the following special characters:

```
* ? / \ | . , ; : + = < > [ ] ( ) & ^
```

If no label is specified, then the user is asked for the label. If no label is given, then the current label is simply removed from the diskette or partition, and a new one is not added. If a label is given, then the specified label replaces the old one.

If a drive is specified, then that drive is affected. If no drive is specified, then the default drive is affected.

Example:

The following example labels drive C with the label, MYDISK:

```
C> LABEL C: MYDISK
C>
```

LOADHI Command

Function: Runs a program in upper memory.

Type: Internal

Syntax: LOADHI [*filename*]

Parameters:

filename Name of program to run in upper memory.

Description:

The LOADHI command causes the specified program to be run in upper memory, so that in the event the program is a TSR, it can occupy memory away from the 640 KB main memory area. This leaves more contiguous memory for other applications.

There is no advantage to running a program in upper memory that does not terminate and stay resident in memory..

Example:

The following example runs SMARTDRV in an upper memory block (note that an extended memory manager must be running in order for a UMB to exist):

```
C> LOADHI SMARTDRV
General Software SMARTDRV Disk Cache Version 1.0

C>
```

MEM Command

Function: Displays memory usage statistics.

Type: Internal

Syntax: MEM [/C] [/F] [/S] [/P]

Parameters:

/C	Classifies programs by memory usage.
/F	Displays information about free memory.
/S	Displays system kernel memory pool.
/P	Pauses after each screenful of information.

Description:

The MEM command displays statistics about the quantity and usage of RAM in the running system, including low memory, extended memory, and operating system memory pool.

Examples:

The following example shows a basic MEM display:

```
C> MEM
Memory Type        Total  =   Used  +   Free
---------------  -------   -------   -------
Operating System     57K       12K       45K
Conventional        583K      108K      475K
Upper                 0K        0K        0K
Extended (XMS)      912K        0K      912K
INT 15h Extended      0K        0K        0K
---------------  -------   -------   -------
Total memory       1552K      120K     1432K

Total Under 1MB     640K      120K      520K

Largest executable program size    452K
Largest free upper memory block      0K

The high memory area is available for use.
Largest free XMS block size = 912K.
Number of XMS handles available = 40927.

C>
```

MKDIR Command

Function: Makes a subdirectory.

Type: Internal

Syntax: MKDIR [*d:*]*path*

Parameters:

d:	Drive to create the directory on.
path	Pathname of the directory to be created.

Description:

The MKDIR command (abbreviated MD) creates a subdirectory of a root directory or a subdirectory. By using the MKDIR command, tree-structured file systems can be created that can effectively manage thousands of files by grouping them hierarchically.

If a drive is specified, then the directory is created on the specified drive. Otherwise, it is created on the default drive.

A pathname of the directory to be created must be given. The name must be unique and cannot name an existing file or subdirectory, as duplicate files are normally not supported by file systems.

Examples:

The following example creates a subdirectory of the current directory on the default drive called JOE:

```
C> MKDIR JOE
C>
```

The following command creates a directory called FINANCE in the root directory of drive A:

```
C> MKDIR A:\FINANCE
C>
```

MODE Command

Function: Sets or displays operational modes for devices.

Type: External

Syntax:
MODE LPT*n*:[*pagewidth*][,[*linesperinch*][,P]]
MODE COM*x*:*baudrate*,[*parity*],[*databits*],[*stopbits*],[P | -]
MODELPT*n*:=COM*x*:
MODE *display*

Parameters:

n	Narallel port number (1-3).
x	Asynchronous communications port number (1-4).
pagewidth	Columns per printed line.
linesperinch	Sines per vertical inch.
baudrate	Transmission baudrate for async devices.
parity	Parity type (even, odd, or none).
databits	Data bits per transmitted character.
stopbits	Stop bits per transmitted character.
P	Infinite retry should be enforced.
-	Infinite retry should be terminated.
display	Display mode type to change to.

Description:

The MODE utility program displays the status of devices in the system, or changes their mode of operation. Supported devices are the parallel printers, the asynchronous communications ports, and the displays. If no parameters are specified, then the status of the devices in the system is displayed.

The first form listed above changes the settings for the default pagewidth and number of lines that can be printed in one inch (vertically) associated with a parallel printer. This allows listings to be wrapped correctly on smaller paper. The parallel printers are named LPT1, LPT2, LPT3, and LPT4.

The second form listed above changes the settings for a communications port. The ports are named COM1, COM2, COM3, and COM4. The system initializes each COM port to 2400 baud, even parity, 7 data bits, and 1 stop bit.

The supported baud rates are: 110, 150, 300, 600, 1200, 2400, 4800, 9600, and 19,200.

The supported parity types are: E (even), O (odd), and N (none).

The supported stop bits are 1 or 2. For most communications applications, if the baud rate is 110, then 2 should be selected, otherwise 1 should be used.

If P is specified, then MODE uses infinite retry on the parallel or serial device, so that I/O doesn't time-out due to temporary device problems, such as out-of-paper.

The third form re-routes I/O destined to the specified parallel printer through an asynchronous COM port. This allows printers attached to serial ports to be used transparently.

The fourth form changes the display mode, as follows:

_ MONO	Switches to monochrome (TTL) display.
_ BW40	Switches to black-and-white, 40 column display.
_ BW80	Switches to black-and-white, 80-column display.
_ CO40	Switches to color, 40-column display.
_ CO80	Switches to color, 80-column display.
_ 40	Switches to 40-column display.
_ 80	Switches to 80-column display.

Examples:

The following example shows how a dual-monitor system can switch back-and-forth between two monitors:

```
C> MODE MONO              (switch to monochrome screen)
C> MODE CO80              (switch to color screen)
```

The following example re-routes LPT1 through the second COM port:

```
C> MODE LPT1:=COM2:
```

```
C>
```

The following example disables any rerouting of LPT1:

```
C> MODE LPT1:
C>
```

The following example sets up the second communications port for 9600 baud, no parity, 1 stop bit, and 8 data bits. Infinite retry is installed:

```
C> MODE COM2:9600,N,8,1,P
C>
```

MORE Command

Function: Displays output one screenful at a time.

Type: External

Syntax: MORE < *filepath*
progpath | MORE

Parameters:

filepath	Pathname of a file to be paged through.
progpath	Name of a program to run.

Description:

The MORE utility program is a filter that can be used by itself or in a pipeline (see examples above) to view a file or the output of a program one screenful at a time. After each screenful, MORE pauses and waits for the user to press a key, after which the next screenful of information is displayed.

Examples:

The following examples show how the contents of a file may be typed out, pausing between each screenful of 24 lines. Either command will achieve the same effect:

```
C> TYPE MYFILE.DAT | MORE
```

```
C> MORE < MYFILE.DAT
```

The following example shows how you can combine DIR and MORE to achieve the same effect as the DIR/P command:

```
C> DIR | MORE
```

The following example shows how a fictitious user report generator program might be run, piping the results into the MORE filter.

```
C> MYREPORT | MORE
```

OPEN Command

Function: Opens a text file for reading and writing.

Type: Internal

Syntax: OPEN *file [d:]pathname*

Parameters:

file	Environment variable to be assigned to the file
d:	Drive on which the specified file resides
pathname	Pathname of the file to be opened

Description:

The OPEN command opens an existing file or creates a new file, for reading and writing with the READ and WRITE commands.

COMMAND.COM's private handle to the file is kept in the specified environment variable, and the READ, WRITE, and CLOSE commands accept this environment variable as the local name of the file to read, write, or close.

If the environment variable exists prior to the OPEN, then it is reassigned, causing its previous contents to be destroyed. Thus, using (system) names such as COMSPEC or PATH is not as good an idea as names like MYFILE, PAYROLL, CUSTOMERS, and so on.

Examples:

The following example opens a file called TANGO.DAT, reads a line of text from it into the variable called BOUNTY and then closes the file:

```
C> OPEN TANGO TANGO.DAT
C> READ TANGO BOUNTY
C> CLOSE TANGO
```

PATH Command

Function: Displays or sets the current path.

Type: Internal

Syntax: PATH [*pathlist*]

Parameters:

pathlist List of directory paths to search.

Description:

The PATH command displays or changes the current search path that is used by the command processor, COMMAND.COM, to locate user programs and batch files.

If no pathlist parameter is specified, then the current path is displayed. If a pathlist parameter is specified, then the path will be changed to the one specified.

The current path is actually stored as an environment variable, in the form:

```
PATH=path1;path2;path3;path4;...;pathn
```

where *path1* is the first directory to search for programs and batch files in, *path2* is the next directory, and so on. When COMMAND.COM tries to run a program or batch file, it first looks in the current directory of the default drive, and then looks through all of the named directories in the PATH environment variable until the program or batch file is located.

You may also change your PATH variable with the SET command, documented later in this chapter. Both commands have the same effect, although using the PATH command without operands to display the current path is easier than scanning through the output from the SET command to accomplish the same thing.

Examples:

The following example shows the current path:

```
C> PATH
PATH=C:\DOS;D:\UTILS
C>
```

The following example sets the current path to first look in the DOS directory of drive C, and then in the PROGRAMS directory on the default drive:

```
C> PATH C:\DOS;\PROGRAMS
C>
```

PAUSE Command

Function: Pauses to allow for user intervention.

Type: Internal

Syntax: PAUSE [*message*]

Parameters:

message An optional message to be displayed before pausing.

Description:

The PAUSE command is typically used in batch files to suspend execution of the batch file, print a message on the screen, and wait for the user to press a key after some action has been performed. This is useful in instances where, for example, diskettes must be changed before continuing.

PAUSE displays the following message on the screen before accepting a keypress from the user:

```
Strike any key when ready . . .
```

Example:

The following example prints a message and waits for the user to press a key before continuing:

```
C> PAUSE Insert your disk in drive A now.
Insert your disk in drive A now.
Strike any key when ready . . . _
C>
```

PROMPT Command

Function: Changes the current command shell prompt.

Type: Internal

Syntax: PROMPT [*string*]

Parameters:

string Prompt string to be used by COMMAND.COM.

Description:

The PROMPT command maintains the PROMPT environment variable that is used by COMMAND.COM to display something before the user is asked to type-in a command. The prompt string contains metacharacters that form a simple language, shown in a table below.

The default PROMPT variable is ng. This has the effect of showing the current drive letter followed by a "greater-than" sign:

```
A>
```

If no string is specified, then the prompt is reset to the default prompt, above. Otherwise, the prompt string is changed for the next command prompting.

The metacharacters supported by EMBEDDED DOS 6-XL are the same ones that are supported by MS-DOS. They are shown in Table 5.1:

Metacharacter	Displayed as . . .
$$	Dollar sign
$t	System time
$d	System date
$p	Current path of default drive
$v	MS-DOS compatibility version number
$n	Default drive letter without the colon
$g	Greater-than sign (>)
$l	Less-than sign (<)
$b	Vertical bar (|)
$q	Equal sign (=)
$e	Escape character (0x1b)
$_	Carriage-return/Line-feed pair
$i	Task bar at top of screen

Table 5.1. Metacharacters supported by PROMPT.

Example:

The following example changes the prompt to display the date and time on one line, and then the current directory on the following line:

```
C> PROMPT $d $t $_$p $g
```

READ Command

Function: Reads a line of input from an OPENed file.

Type: Internal

Syntax: READ *file buffer*

Parameters:

file Environment variable assigned to file at OPEN time
buffer Environment variable to read the data into

Description:

The READ command reads the next sequential line-feed-delimited line of text from the specified file opened with the OPEN command. The data is stored in the specified environment variable used as a buffer.

If the *buffer* environment variable did not exist prior to the READ command's execution, then it is created.

If the file is at EOF, a zero-byte buffer is read.

Examples:

The following example opens a file called TANGO.DAT, reads a line of text from it into the variable called BOUNTY and then closes the file:

```
C> OPEN TANGO TANGO.DAT
C> READ TANGO BOUNTY
C> CLOSE TANGO
```

REM Command

Function: Provides a remark in a batch file.

Type: Internal

Syntax: REM [*remark*]

Parameters:

remark Any optional comment in any format whatsoever.

Description:

The REM command provides a simple way of entering a free-form comment in a batch file, that has no side effects. The REM command may be used interactively, but has no effect.

Example:

```
C> REM this is a remark in a batch file!
C>
```

RENAME Command

Function: Renames a file or group of files.

Type: Internal

Syntax: REN[AME] [*d:*]*path newpath*

Parameters:

d:	Drive on which files are to be renamed.
path	Wildcarded pathname of file(s) to be renamed.
newpath	Wildcarded new pathname of files.

Description:

The RENAME command (abbreviated REN) renames a file or group of files. Files cannot be moved in the directory structure with this command; instead, only their filenames are altered within the directory in which they reside.

Wildcards may be used in the second pathname to indicate that the characters in that component of the first filename are to be kept as-is. For example, to rename the file JANUARY.RPT to JANUARY.SAV, you could use the following command to avoid typing JANUARY twice:

```
C> RENAME JANUARY.RPT *.SAV
C>
```

Example:

The following example renames all of the files with .WKS extensions in the current directory of the default drive to have .BAK extensions instead:

```
C> REN *.WKS *.BAK
C>
```

REWIND Command

Function: Rewinds a file to its beginning.

Type: Internal

Syntax: REWIND *file*

Parameters:

file Environment variable assigned to file at OPEN time

Description:

The REWIND command rewinds a file that was previously opened with the OPEN command. The effect of this command is to reposition the file pointer to the beginning of the file, so that the next record read from the file will be the first one in the file. Similarly, the next WRITE command will overwrite the first record in the file.

Examples:

The following example opens a file called TANGO.DAT, reads a line of text from it into the variable called BOUNTY, rewinds the file, and rewrites the record with a different record content:

```
C> OPEN TANGO TANGO.DAT
C> READ TANGO BOUNTY
C> REWIND TANGO
C> SET NEWREC=This is a new record
C> WRITE TANGO NEWREC
```

RMDIR Command

Function: Removes a subdirectory.

Type: Internal

Syntax: RMDIR [*d:*]*path*

Parameters:

d:	Drive to remove the directory from.
path	Pathname of the directory to be removed.

Description:

The RMDIR command (abbreviated RD) removes a subdirectory of a root directory or of a subdirectory. This command can only be used to delete directories, and cannot be used to delete files, even if they are inside the directory to be removed. Conversely, the DEL command cannot delete directories; only the files they contain.

If a drive is specified, then the directory on the specified drive is removed. Otherwise, the default drive is assumed. A pathname of the directory to be removed must be given.

Examples:

The following example removes a subdirectory of the current directory on the default drive called JOE:

```
C> RMDIR JOE
C>
```

The following command removes a directory called FINANCE in the root directory of drive A:

```
C> RMDIR A:\FINANCE
C>
```

SET Command

Function: Displays or changes the environment strings.

Type: Internal

Syntax: SET [*variable*=[*string*]]

Parameters:

variable	Variable name to change the value of.
string	String to be assigned to the variable.

Description:

The SET command displays the entire environment space (one variable per line), or changes the assignment of one variable in the environment space.

If no operands are specified, then the SET command simply displays all of the environment variables in the environment space.

If a variable name and an equal sign is given, but no string is specified, then the variable name is removed from the environment space. If the string is specified, then the previous definition of the variable is deleted, and the new one is installed in the environment.

Common commercial software packages use the environment space to hold their configuration parameters. Batch files may also access the environment variable assignments with the following syntax:

%*variable*%

The use of percent signs around a text string tells COMMAND.COM to substitute the actual string assignment of the variable name with the variable in percents.

When a program is run, the current environment is cloned and the program runs with its own copy of the environment space. Thus, if it changes any variable values in its copy of the environment space, it will not affect the environment space of the command interpreter when the program terminates.

The size of the environment space may be specified when invoking the COMMAND.COM command processor. See the COMMAND utility program documentation in this chapter for details.

Examples:

The following command sets the PROMPT variable to a new string that displays the current directory and a '>' sign:

```
C> SET PROMPT=$p$g
C:\PROGRAMS>_
```

SHIFT Command

Function: Allows access to multiple batch file arguments.

Type: Internal

Syntax: SHIFT

Parameters:

none.

Description:

The SHIFT command shifts the contents of the 9 batch file arguments so that %2 is copied into %1, %3 is copied into %2, and so on. The %9 variable is reset to the empty string.

SMARTDRV Command

Function: Disk cache utility.

Type: External

Syntax: SMARTDRV [/X] [/T:*n*]

Parameters:

/X	Disables write-through for floppy drives.
/T:*n*	Specifies number of kilobytes of XMS memory to use for cache.

Description:

The SMARTDRV utility program is a disk cache subsystem that installs as a TSR and remains resident in memory for the remainder of the session. Once installed, it optimizes file system performance by caching disk blocks in XMS memory.

SMARTDRV requires XMS in order to operate properly. Therefore, you must load HIMEM.SYS from CONFIG.SYS before running SMARTDRV.

Examples:

It is convenient to run SMARTDRV directly from CONFIG.SYS. The following example shows how:

```
INSTALL=C:\SMARTDRV.EXE
```

SORT Command

Function: Sorts lines of text.

Type: External

Syntax: SORT [/R] [/+*n*]

Parameters:

/R	Input is to be sorted in reverse order.
/+*n*	Starting column of field to sort on.

Description:

The SORT utility program is a filter that reads lines of text from standard input, sorts them, and writes the sorted file to standard output. Piping or redirection is needed to make use of this utility.

If the /R option is specified, then a descending rather than ascending sort is performed.

If the /+n option is specified, then the lines will be sorted on the field starting at column 'n' and ending at the last column, of each line read from standard input. By default, the file is sorted on column 1.

The ability of SORT to handle large files is dependent on the amount of conventional memory available at the time SORT runs.

Examples:

The following example sorts a directory listing:

```
C> DIR | SORT
```

The following command sorts a report by column 27:

```
C> SORT /+27 < JANUARY.RPT
```

SWITCH Command

Function: Displays or changes the option switch character.

Type: Internal

Syntax: SWITCH [*character*]

Parameters:

character Switch character to be used.

Description:

The SWITCH command allows the user to display or change the option switch character. The default value for this character is the forward slash (/), and is the one documented as the switch separator in all of the commands in this chapter.

For compatibility with other operating systems, this character can be changed to some other character (for example, a minus sign). Because all of the utility programs use the operating system's switch character rather than a hard-coded slash character, they will all function with whatever switch character the user selects.

If no character is specified, then the current switch character is displayed.

Examples:

The following example displays the current switch character:

```
C> SWITCH
SWITCH is '/'.
C>
```

The following command changes the switch character to a minus sign:

```
C> SWITCH '-'
C>
```

SYNCH Command

Function: Flushes all system file buffers for orderly shutdown.

Type: Internal

Syntax: SYNCH

Parameters:

none.

Description:

The SYNCH command provides a synchronization checkpoint feature that enables a batch file to flush the file system's buffers to disk before doing something that would otherwise cause a disorderly shutdown (such as executing a PARK program that parks the disk heads, or physically turning it off.)

After the SYNCH command executes, all buffers in the file system are guaranteed to be flushed until the next write, caused by opening, closing, writing, or otherwise affecting any file name space in the system.

SYNCH can only be used to synchronize local file systems, and may or may not flush buffers on other workstations in a distributed file system.

Examples:

The following example flushes the file system so that the workstation can be turned off:

```
C> SYNCH
C> (the workstation can now be turned off)
```

SYS Command

Function: Installs EMBEDDED DOS 6-XL system files on a drive.

Type: External

Syntax: SYS *d:*

Parameters:

d: Drive to install system files on.

Description:

The SYS utility program copies the two system files, DOS.SYS and COMMAND.COM, to the root directory of the target drive. These files are loaded from the directory specified in the COMSPEC environment variable.

The SYS utility also installs a new partition boot record on the drive so that it can boot the DOS.SYS file instead of MS-DOS system files.

Example:

The following example installs the system files on drive C:

```
A> SYS C:
Copying system files...
System transferred.
A>
```

TIME Command

Function: Displays or changes the system time.

Type: Internal

Syntax: TIME [*hh:mm:ss*]

Parameters:

hh:mm:ss System time in hours, minutes, and seconds.

Description:

The TIME command displays or changes the system time. If no parameter is specified, then the current system time is displayed, and the user is queried for the new system time. If the user just presses the ENTER key, the system time is not changed. If the user enters a new time, then the system's real-time-clock is updated.

If a parameter is specified, then it must be in the form: hh:mm:ss. It is acceptable to omit the seconds. The hours, minutes, and seconds are checked by TIME to ensure that they are reasonably correct. Thus, it is invalid to enter the time: 99:88:77, since hours cannot be larger than 23, minutes cannot be greater than 59, and seconds cannot exceed 59.

Example:

The following example displays the current time and changes it interactively:

```
A> TIME
Current time is  2:12:14.16
Enter new time: 13:12:14
A>
```

TREE Command

Function: Displays directory structure of drive.

Type: External

Syntax: TREE [*d:*] [/F]

Parameters:

d:	Drive to display tree for.
/F	Displays filenames as well as directory names.

Description:

The TREE utility program displays the directory structure of a drive by recursively descending into each subdirectory, until all of the directories are listed.

If the /F option is specified, then filenames of files in the directories are displayed as they are traversed.

If no drive letter is specified, then the default drive is assumed; otherwise, the specified drive is scanned.

Example:

The following example displays the directory structure for drive D:

```
A> TREE D:
```

TYPE Command

Function: Displays the contents of a file.

Type: Internal

Syntax: TYPE [*d:*]*path*

Parameters:

d:	Drive that the file resides on.
path	Pathname of the file to display.

Description:

The TYPE command copies the contents of the specified file to standard output (usually, the screen). If the drive letter is not specified, then the default drive is assumed.

Example:

The following example displays the contents of the CONFIG.SYS file on drive C:

```
A> TYPE C:\CONFIG.SYS
FILES=20
BUFFERS=20
A>
```

VER Command

Function: Displays the version information about the operating system.

Type: Internal

Syntax: VER

Parameters:

none.

Description:

The VER command displays the MS-DOS emulation version number, as well as the version of the EMBEDDED DOS 6-XL operating system that is running.

VERIFY Command

Function: Displays or changes the status of the VERIFY flag.

Type: Internal

Syntax: VERIFY [ON | OFF]

Parameters:

ON	Disables lazy-writing in the file system & disk drivers.
OFF	Enables lazy-writing in the file system & disk drivers.

Description:

The VERIFY command changes or displays how EMBEDDED DOS 6-XL handles I/O to disk files and directory structures. If VERIFY is ON, then EMBEDDED DOS 6-XL verifies immediately that disk I/O is completed successfully before telling the user that it was. This is accomplished by writing data directly to disk, without temporarily storing it in a file system or disk driver cache.

If VERIFY is OFF, then EMBEDDED DOS 6-XL caches writes to files and defers the actual writing to disk, enabling multiple writes to the same sectors to be served much faster. The cache is automatically written to disk in the background during "dead time", when the disk is not busy. This is accomplished with the multitasking threads and semaphores that the EMBEDDED DOS 6-XL kernel supports.

Examples:

If no parameters are specified on the VERIFY command, then the status of the VERIFY flag is displayed. For example:

```
C> VERIFY
VERIFY is ON.
C>
```

If ON or OFF is specified, then the VERIFY flag is set to that value. For example:

```
C> VERIFY OFF
C> VERIFY
VERIFY is OFF.
C>
```

VOL Command

Function: Displays the volume label on a drive.

Type: Internal

Syntax: VOL [*d:*]

Parameters:

d: Drive or partition to display the volume label for.

Description:

The VOL command displays the volume label of a diskette or a hard disk partition, much like the external LABEL utility program allows. VOL does not allow the user to change the volume label.

If the drive letter is not specified, then the default drive is assumed.

WRITE Command

Function: Writes a line of text to a file opened with the OPEN command.

Type: Internal

Syntax: WRITE *file buffer*

Parameters:

file Environment variable assigned to file at OPEN time
buffer Environment variable containing data to write

Description:

The WRITE command sequentially writes a line of text to the specified file previously opened with the OPEN command. The data in the *buffer* environment variable is written to the file. A carriage-return and line-feed are automatically appended to the line of data.

Examples:

The following example opens a file called TANGO.DAT, writes a line of text to it from the variable called BOUNTY and then closes the file:

```
C> OPEN TANGO TANGO.DAT
C> SET BOUNTY=This is a line of text to write.
C> WRITE TANGO BOUNTY
C> CLOSE TANGO
```

XCOPY Command

Function: Copies groups of files and/or subdirectories.

Type: External

Syntax: XCOPY *source* [*destination*] [/A | /M] [/P] [/S] [/E] [/V] [/W]

Parameters:

source Specifies the file(s) to be copied
destination Specifies the location of the copies
/A Copies files with the archive bit set without changing the bit
/M Copies files with the archive bit set, clearing while copying
/P Prompts before copying each file
/S Copies subdirectories except empty ones
/E Copies subdirectories including empty ones
/V Verifies each new file
/W Prompts for a key press before copying the files

Description:

The XCOPY command provides enhanced functionality over the COPY command supported by COMMAND.COM. It allows the copying of whole directory trees of files, and makes use of XMS memory (provided by HIMEM.SYS) to buffer the files, thereby optimizing performance.

Examples:

The following example copies the entire contents of a diskette in drive A: to a directory called DISK on drive C:

```
C> XCOPY  A:*.*  C: /E
```

Chapter 5

WRITING REAL-TIME APPLICATIONS

This chapter describes how to write EMBEDDED DOS 6-XL application software that uses the real-time kernel features, and discusses the design considerations that apply to writing multithreaded applications using DOS tools.

If you are writing DOS applications that do not use EMBEDDED DOS 6-XL features such as multithreading, then this chapter may be skipped.

The EMBEDDED DOS 6-XL APIs (there are many Application Program Interfaces, or APIs, in EMBEDDED DOS 6-XL) are callable from assembly language through software interrupts, or from high-level languages using large model C calling conventions through interface modules found in the GENERAL\LIB directory.

For examples on how to call the system services and build multitasking applications for EMBEDDED DOS 6-XL, see the programs (including source and a MAKEFILE) in the GENERAL\EXAMPLES directory.

Using Compilers, Linkers, and Assemblers

Before starting to use kernel, executive, and other API services, we recommend that you learn about how to use your development tools to write code that can run in a multithreaded environment and interact with the services.

The most important issues are writing large model code, no assumptions about SS==DS, and no stack probes.

You may wish to use the following recommended switches when using Borland C++:

```
tasm /DBORLAND /m2 /ml /t /UM510 myprog.asm

bcc  -c -dBORLAND -ml -w- -a- myprog.c

tlink -d -c -m c0l.obj myprog.obj
```

The following switches are recommended for the Microsoft MASM and Visual C++ environment:

```
masm /Ml /T myprog;

cl /Alfu /Gs /Od /Zp1 /Lr /c myprog.c

link /MAP /SEGMENTS:512 myprog;
```

Below is a sample piece of the standard MAKEFILE we use in most EMBEDDED DOS 6-XL component directories to build assembler and C software modules. It uses .IFDEF/.ELSE/.ENDIF syntax, which is only available with the General Software GSMAKE utility.

```
#       Set BORLAND=anything in the ENVIRONMENT SPACE with the SET
#       command to enable the BORLAND BCC & TASM build.  If this is
#       disabled, Microsoft MSC and MASM are chosen by default.

.IFDEF BORLAND

ASM=    tasm
AFLAGS= /DBORLAND /DDEBUG=0 /m2 /ml /t /UM510
LIBS=   cl.lib
C=      bcc
CFLAGS= -c -DBORLAND -ml -nOBJ\ -w- -a-
LINK=   tlink
LFLAGS= -d -c -m c0l.obj
LFLAGSTSR= -d -c -m

.ELSE

LINK=   link
LIBS=
LFLAGS= /MAP /SEGMENTS:512
C=      cl
CFLAGS= /AL /Gs /Od /Zp1 /FoOBJ\ /Lr /c
ASM=    masm
AFLAGS= /Ml /T

.ENDIF
```

We suggest that you start with applications that have been prewritten to show how EMBEDDED DOS 6-XL services are used. Try compiling the sample programs in GENERAL\EXAMPLES and running them. Make sure you have a good understanding from the comments in the programs of what they are supposed to do and how they interact with the kernel and executive APIs.

Then, add clone one of the example programs to a new name, and add the new program to the MAKEFILE in GENERAL\EXAMPLES. Edit the program as needed to try out other functions or programming methods. Learning by example is a quick way to become familiar with the new services you will be using in your production code.

DOS INT 21h Functions

You can use any INT 21h functions supported by generic DOS in EMBEDDED DOS 6-XL programs. This API is reentrant, so you can call DOS from several threads simultaneously, and EMBEDDED DOS 6-XL will process the requests concurrently.

You can also call the INT 21h functions at interrupt time, but this practice can lead to unsafe designs where a thread issues an INT 21h file I/O call on a handle, and then an interrupt service routine is executed which also performs INT 21h file I/O on the same handle. The result is that the ISR blocks to wait for the thread to finish, but the entity that gets blocked is the current thread, which will effectively block on itself. To avoid this potential deadlock, spawn threads from ISRs to do the work at task time instead of interrupt time. That way, blocking occurs in the context of an independently-blockable thread.

INT 21h functions may be called from assembly language or a high-level language such as C. To call from assembly language, set registers to values specified in your INT 21h reference guide and execute an INT 21h instruction. To call from C, use the _intdos or _intdosx functions (these are provided by your C Library vendor; if they don't provide them, link with the GENERAL\LIB\OBJ\INTDOS.OBJ module, which provides access to INT 21h functionality).

C Runtime Library Functions

In general, virtually all of the C library functions supplied with your Borland or Microsoft compiler will work in the EMBEDDED DOS 6-XL environment. There are precautions you must take to ensure that they are used properly in a multitasking environment, however.

The three biggest issues surrounding use of C library functions are reentrancy, stack probes, and assumptions about DS==SS. If you are not writing a multithreaded application, then these issues do not have to be addressed in your application.

If you are writing multithreaded applications (those that allocate additional threads or timers), then you need to be concerned with reentrancy of the code that will be run by multiple threads or by system timers or ISRs. Most C library functions are not designed to be reentrant. They use static variables in their DATA segment to hold temporary values, and these temporary data cannot be used by two threads at once; hence, the nonreentrant nature of these functions. Examples are *printf*, *sprintf*, *fopen*, *fclose*, *fread*, *fwrite*, and so on. Examples of functions that are usually reentrant are *strcpy*, *strcmp*, and *atoi*.

The second most important issue surrounding the use of C library functions in a multithreaded application is the problem of stack probes being inserted by the compiler into the initialization code of each compiled function. Ordinarily, a function creates room on the stack for variables using the following basic sequence of instructions:

```
PUSH    BP
MOV     BP, SP
SUB     SP, xxx
```

However, the stack probe mechanism generates code to call a function called __chkstk, found in the C runtime library, that not only performs the SUB instruction, but it verifies that the values

in SS and SP are reasonable, and that there is enough room on the stack to allocate the variables. This initialization code is shown below:

```
PUSH    BP
MOV     BP, SP
CALL    __CHKSTK
```

This mechanism works fine for the thread that runs the *main* function, but not for additional threads allocated by the user, since they run on independent stacks allocated by the kernel.

While stack probes can be disabled with a compiler switch (in Microsoft C compilers this switch is "/Gs"), you may not have control over your C library routines, which come precompiled by your compiler vendor.

A third major consideration when writing multithreaded applications is that, since each thread runs on its own independent stack, its SS register is necessarily different from other threads' SS registers, and therefore it isn't possible for all the threads' SS registers to be equal to the DGROUP value normally maintained in the DS register.

Some compilers attempt to optimize their instruction sequences by eliminating segment loads through an assumption that the SS register contains the same value that would be stored in DS, so they refer to items in the data segment through SS overrides. This optimization is short-sighted and doesn't work in multitasking environments.

You can compile your own code without this assumption (the Microsoft switches to cause this assumption to be avoided is "/Alfw"), but you don't have control over your C runtime library functions' compilation unless you recompile it yourself with the proper switches.

Not all C library functions have this problem, and vendors are moving to eliminate this sort of optimization in their libraries.

3rd-Party Library Functions

The same concerns for the C library functions apply to third-party library functions in multitasking applications:

- _ Reentrancy controls
- _ Elimination of stack probes
- _ No assumptions about DS==SS

For ways of addressing these issues, see the previous section about using C library functions in a multitasking environment.

In addition to these issues, the third-party library you are using may program external devices and use interrupt service routines that were tested under a single-tasking environment, but may fall apart under the stress of a multithreaded environment.

For example, if you are using a communications library that supports interrupt-driven RS-232 communications, the ISR that handles communication interrupts may not be written with the idea that it could be preempted when the scheduler determines that the current thread has exhausted its timeslice. Scheduling the CPU away from the ISR could cause characters to be lost. To solve ISR priority problems, you can use the IRQPRIORITY= command in CONFIG.SYS to change task-time scheduling to operate at a lower priority than your device interrupt handlers.

In general, we won't be able to anticipate all of the potential problems that you might have with third-party libraries, but the more you know about the operation of EMBEDDED DOS 6-XL, the more qualifying questions you can ask third-party vendors when choosing auxilliary packages to work with your EMBEDDED DOS 6-XL application.

EMBEDDED DOS 6-XL Kernel Services

The multitasking services of EMBEDDED DOS 6-XL are accessed through the kernel API, documented in the Technical Reference Manual. These services provide access to kernel objects, including threads, timers, events, mutexes, system pool, named objects, and multiprocesor spinlocks. This section describes how to access the kernel services from your application program.

Calling Kernel Functions from Assembly Language

Kernel services are defined for assembly language programs in the header file, INC\KERNEL.INC. The kernel software interrupt is 2dh, and the interface passes operands in registers. The following program is a simple example that calls the *PassTimeSlice* kernel function. Note that the example doesn't really exhibit any new behavior, because it just passes its timeslice and gets rescheduled again, but this illustrates how to set-up an assembly language program to call the kernel functions:

```
        TITLE   SIMPLE - Simple Interface Test Program.

        include ..\inc\usegs.inc
        include ..\inc\udefines.inc
        include ..\inc\umacros.inc
        include ..\inc\ustruc.inc

        include ..\inc\kernel.inc

        include ..\inc\dosapi.inc
        include ..\inc\doserr.inc

;       Define the stack.

_STACK  SEGMENT
        db      512 dup ('$')
TopStack =      $
_STACK  ENDS

UCODE   SEGMENT
```

```
        ASSUME  CS:CGROUP, DS:NOTHING, ES:NOTHING, SS:NOTHING
DefProc Main, PUBLIC, FAR
        mov     ax, DGROUP
        mov     ds, ax
        ASSUME  DS:DGROUP               ; (DS) = DGROUP.

;       Execute a PASS TIME SLICE kernel function.  The function code
;       is passed through the DL register.  Other operands are passed
;       in other registers according to the specifications in the
;       EMBEDDED DOS 6-XL Technical Reference Manual.

        mov     dl, SYS_PASS_TIME_SLICE ; (DL) = function code.
        int     SYSINT                  ; same as INT 2dh.

;       Now terminate the program, returning to EMBEDDED DOS 6-XL.

        mov     ah, DOSEXIT
        mov     al, 0                   ; successful status code.
        int     21h                     ; terminate program.
EndProc Main

UCODE   ENDS
        END     Main
```

Calling Kernel Functions from C

Kernel services are defined for C language programs in the header file, INC\KERNEL.H. This API is actually provided by a helper module in GENERAL\LIB\KERNEL.LIB, which loads the arguments passed to the C-callable functions on the stack into CPU registers and invokes software interrupt INT 2dh. The following program is a simple example that calls the *PassTimeSlice* kernel function. Note that the example doesn't really exhibit any new behavior, because it just passes its timeslice and gets rescheduled again, but this illustrates how to set-up a C language program to call the kernel functions:

```
/*
// PROGRAM NAME:  SIMPLEC.C.
//
// FUNCTIONAL DESCRIPTION.
//    This program is a simple example of how the kernel functions
//    can be called from large-model C applications.  The program
//    doesn't do anything but make a call to the PassTimeSlice function
//    in the GENERAL\LIB\OBJ\KERNEL.OBJ module, which in turns calls
//    the EMBEDDED DOS 6-XL operating system through INT 2dh.
*/

#define VOID void
#define UCHAR unsigned char
#define USHORT unsigned short
#define ULONG unsigned long
#define FALSE 0
#define TRUE (!FALSE)
```

```
#include <stdio.h>

#include "..\inc\kernel.h"    // kernel API definitions.

VOID main (int argc, char *argv[])
{
    printf ("This is a simple C program that calls a kernel function.\n");

    PassTimeSlice ();         // call the kernel function.

    exit (0);                 // successfully return to EMBEDDED DOS 6-XL.
} // simplec.c
```

To compile and link the example C language program with the EMBEDDED DOS 6-XL kernel interface module, GENERAL\LIB\KERNEL.LIB, use a procedure similar to the following (Microsoft tools are shown, but Borland tools will work equally well):

```
cl /Alfu /Gs /Od /Zp1 /Lr /c simplec.c

link simplec.obj,simplec.exe,,\general\lib\kernel.lib;
```

Calling Kernel Functions from Other HLLs

You may also call EMBEDDED DOS 6-XL kernel functions from other languages, such as Ada, Basic, Pascal, or Modula-2. To accomplish this, you will need to clone the KERNEL.ASM module in GENERAL\LIB to another module name of your choice, edit it to support the calling conventions of your language, and link its corresponding OBJ file with your application.

If your language supports Microsoft C-language calling conventions, then you may link directly with the provided module in GENERAL\LIB\KERNEL.LIB.

Calling I/O Helper Functions

The EMBEDDED DOS 6-XL I/O Helper API is available to application programs written in any language. I/O Helper services are defined in the EMBEDDED DOS 6-XL Technical Reference Manual. These services are used by programs that need access to I/O services without going through the MS-DOS INT 21h services (and thereby avoiding DOS-oriented critical error handling, 20 handle-per-process limitation, and PSP-relative handle management). Normally, programs such as file servers or TSRs that must perform I/O independent of the running programs in the foreground would use I/O Helper services to communicate directly with the EMBEDDED DOS 6-XL I/O system.

Calling I/O Helper Functions From Assembly Language

To gain access to the I/O Helper services from assembly language, you should call the SYS_GET_IOHELP_ADDRESS kernel function to obtain the address of the system's I/O Helper function routine. This function address is saved in a data segment as a DWORD and can be used to call the system's I/O Helper function by a CALL DWORD PTR [IoHelper] instruction. Note that this service is only provided for real-mode operation.

The following example shows how to get the I/O Helper function address and call the I/O Helper services to open a file, write a message to the file, and close it without making INT 21h calls to manipulate the file. The INT 21h calls in this program are used to display status and error messages.

```
        TITLE   IOHELPER - Simple I/O Helper Interface Test Program.

        include ..\inc\usegs.inc
        include ..\inc\udefines.inc
        include ..\inc\umacros.inc
        include ..\inc\ustruc.inc

        include ..\inc\kernel.inc
        include ..\inc\ioapi.inc
        include ..\inc\dosapi.inc
        include ..\inc\doserr.inc

;       Define the stack.

_STACK  SEGMENT
        db      512 dup ('$')
TopStack =      $
_STACK  ENDS

UDATA   SEGMENT

IoHelper dd     ?               ; 16:16 I/O Helper address.

FileName db     'foo', 0        ; name of file to create.

DataBuf db      'This is the data to be written to the file.', 13, 10
BUFSIZE =       ($-OFFSET DGROUP:DataBuf)

SuccessMsg db   'File created, written, and closed successfully.', 13, 10, '$'
NoCreateMsg db  'Unable to create file FOO.', 13, 10, '$'
NoWriteMsg db   'Unable to write to file FOO.', 13, 10, '$'
NoCloseMsg db   'Unable to close file FOO.', 13, 10, '$'

UDATA   ENDS

UCODE   SEGMENT

        ASSUME  CS:CGROUP, DS:NOTHING, ES:NOTHING, SS:NOTHING
DefProc Main, PUBLIC, FAR
        mov     ax, DGROUP
        mov     ds, ax
        ASSUME  DS:DGROUP               ; (DS) = DGROUP.

;       Obtain the system I/O Helper function address.

        mov     dl, SYS_GET_IOHELP_ADDRESS
        int     SYSINT                  ; (DX:AX) = FWA, IoHelp dispatch.
        mov     IoHelper.hi, dx
```

```
        mov     IoHelper.lo, ax         ; save the address for our I/O calls.

;       Now open a file called FOO.

        mov     di, IOHELP_CREATE       ; (DI) = function code.
        mov     dx, ds
        mov     ax, OFFSET DGROUP:FileName ; (DX:AX) = FWA, filename.
        mov     bx, 0                   ; (BX) = 0, meaning create a file.
        mov     cx, 0                   ; (CX) = file attributes.
        call    dword ptr [IoHelper]    ; (BX) = resulting system handle.
        jc      Main_NoCreate           ; if we couldn't create the file.

;       We have created FOO and have a handle in (BX).  Now write to it.

        mov     di, IOHELP_WRITE        ; (DI) = function code.
        mov     dx, ds
        mov     ax, OFFSET DGROUP:DataBuf ; (DX:AX) = FWA, buffer to write.
        mov     cx, BUFSIZE             ; (CX) = # bytes to write.
        call    dword ptr [IoHelper]    ; (CX) = # bytes actually written.
        jc      Main_NoWrite            ; if we couldn't write to the file.

;       We have written to the file, so close it now.

        mov     di, IOHELP_CLOSE        ; (DI) = function code.
        call    dword ptr [IoHelper]    ; close the file.
        jc      Main_NoClose            ; if we couldn't close the file.

;       Now terminate the program, returning to EMBEDDED DOS 6-XL.

        mov     ah, DOSCONSTROUTPUT     ; (AH) = output string function.
        mov     dx, OFFSET DGROUP:SuccessMsg
        int     21h

Main_Exit:
        mov     ah, DOSEXIT
        mov     al, 0                   ; successful status code.
        int     21h                     ; terminate program.

;       We were unable to create the file.

Main_NoCreate:
        mov     ah, DOSCONSTROUTPUT     ; (AH) = output string function.
        mov     dx, OFFSET DGROUP:NoCreateMsg
        int     21h
        jmp     Main_Exit               ; terminate the program.

;       We were unable to write to the file.

Main_NoWrite:
        mov     ah, DOSCONSTROUTPUT     ; (AH) = output string function.
        mov     dx, OFFSET DGROUP:NoWriteMsg
        int     21h
        jmp     Main_Exit               ; terminate the program.
```

```
;       We were unable to close the file.

Main_NoClose:
        mov     ah, DOSCONSTROUTPUT     ; (AH) = output string function.
        mov     dx, OFFSET DGROUP:NoCloseMsg
        int     21h
        jmp     Main_Exit               ; terminate the program.
EndProc Main

UCODE   ENDS
        END     Main
```

Calling I/O Helper Functions From C

To gain access to the I/O Helper services from C, you can make high-level language calls to an I/O Helper interface module, found in GENERAL\LIB\IOHELP.LIB. This module automatically retrieves the address of the system's I/O Helper function when the first I/O Helper function is called, and it hides the details of calling through the resulting DWORD pointer.

The following C language program example shows how to create, write, and close a file using I/O Helper functions through the IOHELP.LIB interface.

```
/*
// PROGRAM NAME:  IOHELPC.C.
//
// FUNCTIONAL DESCRIPTION.
//      This program is a simple example of how to call EMBEDDED DOS 6-XL I/O
//      Helper services from an assembly-language program.  It has no
//      output; it simply opens a file called FOO, writes a message to
//      the file, and closes it again.
//
//      This program must be linked with GENERAL\LIB\OBJ\IOHELP.OBJ in
//      order to resolve the I/O Helper function names.
*/

#define VOID void
#define UCHAR unsigned char
#define USHORT unsigned short
#define ULONG unsigned long
#define FALSE 0
#define TRUE (!FALSE)

#include <stdio.h>

#include "..\inc\system.h"      // general EMBEDDED DOS 6-XL equates.
#include "..\inc\iohelp.h"      // I/O Helper API definitions.

#define BUFSIZE 128
static UCHAR DataBuf [BUFSIZE]; // buffer for file I/O.

VOID main (int argc, char *argv[])
{
```

```
    STATUS retcode;
    HANDLE Handle;
    USHORT Length, BytesWritten;

    printf ("This is a simple C program that creates a file called FOO.\n");

    retcode = IoCreate ("FOO", 0, 0, &Handle);
    if (retcode != DOSERR_SUCCESS) {
        printf ("IoCreate returned failing status, %u.\n", retcode);
        exit (retcode);
    }

    strcpy (DataBuf, "This is the data to be written to the file.");
    Length = strlen (DataBuf);

    retcode = IoWrite (Handle, DataBuf, Length, &BytesWritten);
    if (retcode != DOSERR_SUCCESS) {
        printf ("IoWrite returned failing status, %u.\n", retcode);
        IoClose (Handle);
        exit (retcode);
    }
    if (BytesWritten != Length) {
        printf ("IoWrite only wrote %u bytes; it should have been %u.\n",
                BytesWritten, Length);
        IoClose (Handle);
        exit (-1);
    }

    retcode = IoClose (Handle);
    if (retcode != DOSERR_SUCCESS) {
        printf ("IoClose returned failing status, %u.\n", retcode);
        exit (retcode);
    }

    exit (0);                   // successfully return to EMBEDDED DOS 6-XL.
} // iohelpc.c
```

To compile and link the example C language program with the EMBEDDED DOS 6-XL I/O Helper interface module, GENERAL\LIB\IOHELP.LIB, use a procedure similar to the following (Microsoft tools are shown, but Borland tools will work equally well):

```
cl /Alfu /Gs /Od /Zp1 /Lr /c iohelpc.c

link iohelpc.obj,iohelpc.exe,,\general\lib\iohelp.lib;
```

EMBEDDED DOS 6-XL Executive Services

The executive services of EMBEDDED DOS 6-XL are accessed through the Executive API, documented in the Technical Reference Manual. These services provide access to executive objects, including message ports, and queues. This section describes how to access the executive services from your application program.

Executive services are defined for C language programs in the header files, INC\MESSAGE.H and INC\QUEUE.H. The actual implementation of the message port and queue objects is found in LIB\MESSAGE.C and LIB\QUEUE.C. These objects are not manipulated with software interrupts. Instead, the object implementation code, found in LIB\MESSAGE.LIB and LIB\QUEUE.LIB, is linked with the application program. Multiple programs may link to the same object implementations and communicate through the interfaces; this is handled through named objects in the kernel.

Calling Message Port Functions

The following C language program example shows how to call the message port functions to create a message port, open the port with another handle, write to the message port with one handle, and read from the message port with the other handle. Both handles are then closed, deleting the message port from the system. Message ports offer both synchronous and asynchronous modes; this application uses an asynchronous send and a synchronous receive.

```
/*
// PROGRAM NAME:  MSGPORT.C.
//
// FUNCTIONAL DESCRIPTION.
//      This program is a simple example of how the executive message port
//      functions can be called from large-model C applications.  The program
//      creates a message port, and opens it with another handle, simulating
//      a second program that wishes to communicate with the first one.  Then,
//      a message is written to the first handle, and read from the second
//      handle.  The call to ExSendMsg is necessarily asynchronous so that
//      the ExReceiveMsg call can be executed in the example.
*/

#define VOID void
#define UCHAR unsigned char
#define USHORT unsigned short
#define ULONG unsigned long
#define FALSE 0
#define TRUE (!FALSE)

#include <stdio.h>

#include "..\inc\message.h"     // message port API definitions.

#define MAXBUFLEN 128
static UCHAR Buf [MAXBUFLEN];
static UCHAR OutBuf [MAXBUFLEN];

VOID main (int argc, char *argv[])
{
    HANDLE Handle1, Handle2;
    USHORT len, i, BytesRead;

    printf ("This is a simple C program that uses message ports.\n");

    if (!ExCreateMsgPort ("Message_Server", &Handle1)) {
```

```
        printf ("The Message_Server port could not be created.\n");
        exit (-1);
    }

    if (!ExOpenMsgPort ("Message_Server", &Handle2)) {
        ExCloseMsgPort (Handle1);
        printf ("The Message_Server port could not be opened.\n");
        exit (-1);
    }

    strcpy (Buf, "This is a message.");
    len = strlen (Buf);

    if (!ExSendMsg (Handle1, Buf, len, MSG_FLAGS_BUFFER, 0)) {
        ExCloseMsgPort (Handle1);
        ExCloseMsgPort (Handle2);
        printf ("The message could not be sent.\n");
        exit (-1);
    }

    if (!ExReceiveMsg (Handle2,
                       OutBuf,
                       MAXBUFLEN,
                       &BytesRead,
                       MSG_FLAGS_WAIT,
                       0)) {
        ExCloseMsgPort (Handle1);
        ExCloseMsgPort (Handle2);
        printf ("The message could not be received.\n");
        exit (-1);
    }

    //
    // Compare what we sent with what we received.
    //

    if (BytesRead != len) {
        ExCloseMsgPort (Handle1);
        ExCloseMsgPort (Handle2);
        printf ("The message size was changed.\n");
        exit (-1);
    }

    for (i=0; i<len; i++) {
        if (Buf [i] != OutBuf [i]) {
            ExCloseMsgPort (Handle1);
            ExCloseMsgPort (Handle2);
            printf ("The message was garbled.\n");
            exit (-1);
        }
    }
    printf ("The message was successfully tranmitted & received.\n");

    //
```

```
    // Clean up and return to DOS.
    //

    ExCloseMsgPort (Handle1);   // close the message port handles.
    ExCloseMsgPort (Handle2);
    exit (0);                   // successfully return to EMBEDDED DOS 6-XL.
} // msgport.c
```

To compile and link the example C language program, you will need to link with both the EMBEDDED DOS 6-XL Message Port interface module, GENERAL\LIB\MESSAGE.LIB, and the EMBEDDED DOS 6-XL Kernel interface module, GENERAL\LIB\KERNEL.LIB. Use a procedure similar to the following (Microsoft tools are shown, but Borland tools will work equally well):

```
cl /Alfu /Gs /Od /Zp1 /Lr /c msgport.c

link msgport.obj,msgport.exe,,\general\lib\message.lib+
                              \general\lib\kernel.lib;
```

Calling Queue Functions

The following C language program example shows how to call the queue functions to create a queue, open the queue with another handle, push a datum to the queue with the first handle, pop the datum from the queue with the second handle, and close both handles, deleting the queue from the system. All of the calls are synchronous.

```
/*
// PROGRAM NAME:  QUEUE.C.
//
// FUNCTIONAL DESCRIPTION.
//      This program is a simple example of how the executive queueing
//      functions can be called from large-model C applications.  The program
//      creates a queue and opens it with another handle, simulating a
//      second program that wishes to communicate with the first one.  Then,
//      a data item is pushed to the first handle, and popped from the second
//      handle.
*/

#define VOID void
#define UCHAR unsigned char
#define USHORT unsigned short
#define ULONG unsigned long
#define FALSE 0
#define TRUE (!FALSE)

#include <stdio.h>

#include "..\inc\queue.h"       // queue API definitions.

static ULONG DataToWrite;
static ULONG DataFromQueue;
```

```
VOID main (int argc, char *argv[])
{
    HANDLE Handle1, Handle2;
    USHORT len, i;

    printf ("This is a simple C program that uses queues.\n");

    if (!ExCreateQueue ("My$Queue", &Handle1)) {
        printf ("The My$Queue queue could not be created.\n");
        exit (-1);
    }

    if (!ExOpenQueue ("My$Queue", &Handle2)) {
        ExCloseQueue (Handle1);
        printf ("The My$Queue port could not be opened.\n");
        exit (-1);
    }

    if (!ExPushQueue (Handle1, (PVOID)DataToWrite)) {
        ExCloseQueue (Handle1);
        ExCloseQueue (Handle2);
        printf ("The datum could not be pushed on the queue.\n");
        exit (-1);
    }

    if (!ExPopQueue (Handle2, (PVOID *)&DataFromQueue)) {
        ExCloseQueue (Handle1);
        ExCloseQueue (Handle2);
        printf ("The datum could not be popped from the queue.\n");
        exit (-1);
    }

    //
    // Compare what we sent with what we received.
    //

    if (DataToWrite != DataFromQueue) {
        ExCloseQueue (Handle1);
        ExCloseQueue (Handle2);
        printf ("The datum was garbled.\n");
        exit (-1);
    }
    printf ("The datum was successfully pushed & popped.\n");

    //
    // Clean up and return to DOS.
    //

    ExCloseQueue (Handle1);     // close the queue handles.
    ExCloseQueue (Handle2);
    exit (0);                   // successfully return to EMBEDDED DOS 6-XL.
} // queue.c
```

To compile and link the example C language program, you will need to link with both the EMBEDDED DOS 6-XL Queue interface module, GENERAL\LIB\QUEUE.LIB, and the EMBEDDED DOS 6-XL Kernel interface module, GENERAL\LIB\KERNEL.LIB. Use a procedure similar to the following (Microsoft tools are shown, but Borland tools will work equally well):

```
cl /Alfu /Gs /Od /Zp1 /Lr /c queue.c

link queue.obj,queue.exe,,\general\lib\queue.lib+
                          \general\lib\kernel.lib;
```

Calling Shared Memory Functions

The following C language program example shows how to call the shared memory functions to create a region of shared memory, access the shared memory with a second call, and compare the pointers to see if they point to the same area. All of the calls are synchronous. For API documentation on shared memory services, refer to the EMBEDDED DOS 6-XL Technical Reference Manual.

```
/*
// PROGRAM NAME:  SHAREMEM.C.
//
// FUNCTIONAL DESCRIPTION.
//      This program is a simple example of how the executive shared memory
//      functions can be called from large-model C applications.  The program
//      creates a shared memory object called FOO, writes some data to the
//      shared memory area, then accesses the same memory object by name,
//      compares the data, and deaccesses/deletes the shared memory region.
*/

#define VOID void
#define UCHAR unsigned char
#define USHORT unsigned short
#define ULONG unsigned long
#define FALSE 0
#define TRUE (!FALSE)

#include <stdio.h>

#include "..\inc\shrmem.h"       // shared memory API definitions.

VOID main (int argc, char *argv[])
{
    UCHAR *Buf1, *Buf2;

    printf ("This is a simple C program that uses shared memory.\n");

    if (!ExAllocateSharedMemory ("MY_DATA", 50, &Buf1)) {
        printf ("The MY_DATA shared memory could not be created.\n");
        exit (-1);
    }
```

```
    strcpy (Buf1, "This is the test data.");

    if (!ExAccessSharedMemory ("MY_DATA", &Buf2)) {
        ExDeallocateSharedMemory (Buf1);
        printf ("The MY_DATA shared memory block could not be accessed.\n");
        exit (-1);
    }

    //
    // Compare the two pointers; they should be identical.
    //

    if (Buf1 != Buf2) {
        ExDeaccessSharedMemory (Buf2);
        ExDeallocateSharedMemory (Buf1);
        printf ("The pointers were not the same.\n");
        exit (-1);
    }
    printf ("The shared memory pointers are identical.\n");

    //
    // Clean up and return to DOS.
    //

    ExDeaccessSharedMemory (Buf2);
    ExDeallocateSharedMemory (Buf1);
    exit (0);                   // successfully return to EMBEDDED DOS 6-XL.
} // sharemem.c
```

To compile and link the example C language program, you will need to link with both the EMBEDDED DOS 6-XL Shared Memory interface module, GENERAL\LIB\SHRMEM.LIB, and the EMBEDDED DOS 6-XL Kernel interface module, GENERAL\LIB\KERNEL.LIB. Use a procedure similar to the following (Microsoft tools are shown, but Borland tools will work equally well):

```
cl /Alfu /Gs /Od /Zp1 /Lr /c sharemem.c

link sharemem.obj,sharemem.exe,,\general\lib\shrmem.lib+
                                \general\lib\kernel.lib;
```

Calling Handle Manager Functions

The following C language example shows how to call the executive handle functions to manage shared access to objects managed by the handle manager. All of the calls are synchronous. For API documentation on handle management services, refer to the EMBEDDED DOS 6-XL Technical Reference Manual.

```
/*
// PROGRAM NAME:  HANDLES.C.
//
// FUNCTIONAL DESCRIPTION.
//      This program is a simple example of how the executive handle
```

```
//      management functions can be called from large-model C applications.
//      The program allocates a handle to an object, references the handle,
//      dereferences the handle, and dereferences it again to deallocate it.
*/

#define VOID void
#define UCHAR unsigned char
#define USHORT unsigned short
#define ULONG unsigned long
#define FALSE 0
#define TRUE (!FALSE)

#include <stdio.h>
#include <malloc.h>

#include "..\inc\handle.h"       // handle API definitions.

VOID ObjectReferenceRoutine (PVOID Token)
{
    printf ("The object %08lx has been referenced.\n", Token);
} // ObjectReferenceRoutine

VOID ObjectDereferenceRoutine (PVOID Token)
{
    printf ("The object %08lx has been dereferenced.\n", Token);
} // ObjectDereferenceRoutine

VOID main (int argc, char *argv[])
{
    HANDLE Handle;
    PVOID Table;
    PVOID Object;
    PVOID ObjPtr;

    printf ("This is a simple C program that uses executive handles.\n");

    Table = malloc (100);
    if (Table == NULL) {
        printf ("Unable to malloc 100-byte handle table.\n");
        exit (-1);
    }
    if (!ExCreateHandleTable (Table, 100)) {
        free (Table);
        printf ("Unable to create handle table.\n");
        exit (-1);
    }

    Object = malloc (10);
    if (!ExAllocateHandle (Table,
                           Object,
                           (PHANDLE_REFERENCE_ROUTINE)
                                ObjectReferenceRoutine,
                           (PHANDLE_DEREFERENCE_ROUTINE)
                                ObjectDereferenceRoutine,
```

```
                            &Handle)) {
        ExCloseHandleTable (Table);
        free (Table);
        printf ("Unable to allocate the first handle.\n");
        exit (-1);
    }

    if (!ExAccessHandle (Table,
                         Handle,
                         &ObjPtr)) {
        ExCloseHandleTable (Table);
        free (Table);
        free (Object);
        printf ("Unable to access the object through the handle.\n");
        exit (-1);
    }

    if (Object != ObjPtr) {
        ExCloseHandleTable (Table);
        free (Table);
        free (Object);
        printf ("The wrong object pointer was returned.\n");
        exit (-1);
    }

    ExDeaccessHandle (Table, Handle);   // close ExAccessHandle reference.
    ExDeaccessHandle (Table, Handle);   // close ExAllocateHandle ref.

    //
    // Clean up and return to DOS.
    //

    free (Object);
    ExCloseHandleTable (Table);
    free (Table);
    exit (0);                   // successfully return to EMBEDDED DOS 6-XL.
} // handles.c
```

To compile and link the example C language program, you will need to link with both the EMBEDDED DOS 6-XL Handle Management interface module, GENERAL\LIB\HANDLE.LIB, and the EMBEDDED DOS 6-XL Interrupt Control interface module, GENERAL\LIB\INTCNTL.LIB. Use a procedure similar to the following (Microsoft tools are shown, but Borland tools will work equally well):

```
cl /Alfu /Gs /Od /Zp1 /Lr /c handles.c

link handles.obj,handles.exe,,\general\lib\handle.lib+
                               \general\lib\intcntl.lib;
```

Multitasking

Embedded DOS was designed to support real-time embedded systems applications. Typical requirements of these applications include the following:

- _ Multiple threads or tasks
- _ Priority-based scheduling
- _ Round-robin scheduling within priority groups
- _ Inexpensive (low overhead) timing
- _ Guarding of critical code/data
- _ Inter-thread synchronization
- _ Interaction with interrupt service routines
- _ User-defined thread context (NPX, other hardware)
- _ Object naming

Embedded DOS implements solutions for all of these problems, with a minimal set of kernel functions that define kernel objects. The kernel objects are rigorously defined in the kernel API chapter. Here we will discuss how they are used in practice.

Lightweight Thread Model

Threads are best thought of as virtual CPUs. They have a general register set and a stack, and execute independently of other threads in the system. They can be interrupted by software or hardware interrupts, and are generally unaware of the scheduling process that takes place "behind the scenes" when they are preempted by other threads in the system.

The following C programs shows how to allocate threads to perform work in parallel. Each thread writes characters to the screen using the C Library function *putch*, which is not reentrant (even though the underlying DOS functions are reentrant). To solve this, the program allocates a mutex object to govern access to *putch*, so that it is only called by one thread at any given time.

```
/*
// PROGRAM NAME:  MT1C.C.
//
// FUNCTIONAL DESCRIPTION.
//      This program is a simple example of how to write applications
//      in C that allocate threads to perform work in parallel.  This
//      example illustrates how to:
//
//      1. Allocate & deallocate threads to perform parallel work, and
//      2. Use event objects for synchronization of work, and
//      3. Guard access to critical code such as C library functions
//
//      This program is linked with GENERAL\LIB\OBJ\KERNEL.OBJ to gain
//      access to the kernel API.
*/

#define VOID void
#define UCHAR unsigned char
#define USHORT unsigned short
```

```
#define ULONG unsigned long
#define FALSE 0
#define TRUE (!FALSE)

#include <stdio.h>

#include "..\inc\kernel.h"      // kernel API definitions.

#define MAXCHARS 500            // # chars to output/thread.
#define INVALID 0xffff          // invalid handle.

HANDLE LibraryMutex=INVALID;    // C library mutual exclusion handle.
HANDLE Trigger=INVALID;         // trigger event handle.
HANDLE Done1=INVALID;           // done1 event handle.
HANDLE Done2=INVALID;           // done2 event handle.
HANDLE Thread1=INVALID;         // thread1 handle.
HANDLE Thread2=INVALID;         // thread2 handle.

THREAD _loadds Thread1Func ()
{
    USHORT i;

    //
    // Wait for the main thread to tell us to start.
    //

    WaitEvent (Trigger);

    //
    // Write-out a few '1's on the screen.
    //

    for (i=0; i<MAXCHARS; i++) {
        AcquireMutex (LibraryMutex);
        putch ('1');
        ReleaseMutex (LibraryMutex);
    }

    //
    // Tell the main thread we are finished.
    //

    SetEvent (Done1);

    //
    // Deallocate this thread (the call never returns).
    //

    DeallocateThread ();
} // Thread1Func

THREAD _loadds Thread2Func ()
{
    USHORT i;
```

```
    //
    // Wait for the main thread to tell us to start.
    //

    WaitEvent (Trigger);

    //
    // Write-out a few '2's on the screen.
    //

    for (i=0; i<MAXCHARS; i++) {
        AcquireMutex (LibraryMutex);
        putch ('2');
        ReleaseMutex (LibraryMutex);
    }

    //
    // Tell the main thread we are finished.
    //

    SetEvent (Done2);

    //
    // Deallocate this thread (the call never returns).
    //

    DeallocateThread ();
} // Thread2Func

VOID main (int argc, char *argv[])
{

    printf ("This sample program has a main thread that allocates two\n");
    printf ("worker threads.  Each thread prints its thread number in\n");
    printf ("a loop %u times to show how preemption works in a simple\n",
            MAXCHARS);
    printf ("multithreaded system without prioritization.\n\n");

    AllocateMutex (&LibraryMutex);
    AllocateEvent (&Trigger);
    AllocateEvent (&Done1);
    AllocateEvent (&Done2);

    //
    // Allocate two worker threads and turn them loose together.
    //

    AllocateThread (Thread1Func);
    AllocateThread (Thread2Func);
    SetEvent (Trigger);

    //
    // Wait for both threads to complete.
```

```
    //

    WaitEvent (Done1);
    WaitEvent (Done2);

    //
    // Cleanup now.
    //

    DeallocateMutex (LibraryMutex);
    DeallocateEvent (Trigger);
    DeallocateEvent (Done1);
    DeallocateEvent (Done2);

    exit (0);                   // successfully return to Embedded DOS.
} // mt1c.c
```

This program (mt1c.c) illustrates how threads can be allocated to perform work in parallel, how events are used for synchronization of thread activities, and how mutexes can be used to guard access to critical resources (the C Library functions).

The following program is the assembly language equivalent.

```
        TITLE   MT1 - Multitasking Sample Test Program #1.

;***    MT1 -- Multitasking Sample Test Program #1.
;
;1.     Functional Description.
;       This program shows how to write applications in assembly language
;       that allocate threads to perform work in parallel.  This example
;       shows how to:
;
;       1.  Allocate and deallocate threads to perform parallel work, and
;       2.  Use event objects for synchronization of work, and
;       3.  Clean-up allocated objects using a special INVALID value.

        include ..\inc\usegs.inc
        include ..\inc\udefines.inc
        include ..\inc\umacros.inc
        include ..\inc\ustruc.inc
        include ..\inc\kernel.inc
        include ..\inc\dosapi.inc
        include ..\inc\doserr.inc

;       Define the main thread's stack.

_STACK  SEGMENT
        db      512 dup ('$')
TopStack =      $
_STACK  ENDS

UDATA   SEGMENT
```

```
MAXCHARS =      500

INVALID =       0ffffh

Trigger dw      INVALID         ; trigger event to get threads going.

Thread1 dw      INVALID         ; thread handle #1.
Thread2 dw      INVALID         ; thread handle #2.

Done1   dw      INVALID         ; event set by thread #1.
Done2   dw      INVALID         ; event set by thread #2.

UDATA   ENDS

UCODE   SEGMENT

;***   Thread1Func - Thread Function #1.
;
;   FUNCTIONAL DESCRIPTION.
;       This routine is executed in the context of a separate thread,
;       on a separate stack allocated from system pool.
;
;   MODIFICATION HISTORY.
;       S. E. Jones     93/09/18.       Original.
;
;   WARNINGS.
;       none.
;
;   ENTRY.
;       none.
;
;   EXIT.
;       Does not return.
;
;   USES.
;       all.

        ASSUME  CS:CGROUP, DS:NOTHING, ES:NOTHING, SS:NOTHING
DefProc Thread1Func
        mov     ax, DGROUP
        mov     ds, ax
        ASSUME  DS:DGROUP

;       Wait for the main thread to trigger us to start going.

        mov     ax, Trigger             ; (AX) = trigger event handle.
        mov     dl, SYS_WAIT_EVENT      ; (DL) = function code.
        int     SYSINT                  ; wait before starting I/O.

;       Write '1's to the screen, which will be interspersed with '2's.

        mov     cx, MAXCHARS            ; (CX) = # characters to write.
Thread1Func_Loop:
        mov     ah, 02h                 ; (AH) = output char to console.
```

```
        mov     dl, '1'                 ; (DL) = character to write.
        int     21h
        loop    Thread1Func_Loop        ; write the other characters.

;       Now set our handle to INVALID, so the main thread will not
;       destroy this thread (we will destroy ourselves).

        mov     Thread1, INVALID

;       Set our Done event so the main thread knows we're done.

        mov     ax, Done1               ; (AX) = our done event handle.
        mov     dl, SYS_SET_EVENT       ; (DL) = function code.
        int     SYSINT                  ; set the event.

;       Destroy this thread.

        sub     ax, ax                  ; (AX) = 0 (handle meaning self).
        mov     dl, SYS_DEALLOCATE_THREAD ; (DL) = function code.
        int     SYSINT                  ; destroy this thread.
EndProc Thread1Func

;***   Thread2Func - Thread Function #2.
;
;   FUNCTIONAL DESCRIPTION.
;       This routine is executed in the context of a separate thread,
;       on a separate stack allocated from system pool.
;
;   MODIFICATION HISTORY.
;       S. E. Jones     93/09/18.       Original.
;
;   WARNINGS.
;       none.
;
;   ENTRY.
;       none.
;
;   EXIT.
;       Does not return.
;
;   USES.
;       all.

        ASSUME  CS:CGROUP, DS:NOTHING, ES:NOTHING, SS:NOTHING
DefProc Thread2Func
        mov     ax, DGROUP
        mov     ds, ax
        ASSUME  DS:DGROUP

;       Wait for the main thread to trigger us to start going.

        mov     ax, Trigger             ; (AX) = trigger event handle.
        mov     dl, SYS_WAIT_EVENT      ; (DL) = function code.
        int     SYSINT                  ; wait before starting I/O.
```

```
;       Write '2's to the screen, which will be interspersed with '1's.

        mov     cx, MAXCHARS            ; (CX) = # characters to write.
Thread2Func_Loop:
        mov     ah, 02h                 ; (AH) = output char to console.
        mov     dl, '2'                 ; (DL) = character to write.
        int     21h
        loop    Thread2Func_Loop        ; write the other characters.

;       Now set our handle to INVALID, so the main thread will not
;       destroy this thread (we will destroy ourselves).

        mov     Thread2, INVALID

;       Set our Done event so the main thread knows we're done.

        mov     ax, Done2               ; (AX) = our done event handle.
        mov     dl, SYS_SET_EVENT       ; (DL) = function code.
        int     SYSINT                  ; set the event.

;       Destroy this thread.

        sub     ax, ax                  ; (AX) = 0 (handle meaning self).
        mov     dl, SYS_DEALLOCATE_THREAD ; (DL) = function code.
        int     SYSINT                  ; destroy this thread.
EndProc Thread2Func

;***   Main - Main Entrypoint.
;
;   FUNCTIONAL DESCRIPTION.
;       This routine is the entrypoint of the test program.  We call
;       the kernel function, AllocateThread, to create two threads that
;       run in parallel with the parent thread (us).  The threads make
;       50 DOS calls to output characters to the screen, and then set
;       their event objects so that we know when they have finished.
;
;   MODIFICATION HISTORY.
;       S. E. Jones     93/09/18.       Original.
;
;   WARNINGS.
;       none.
;
;   ENTRY.
;       none.
;
;   EXIT.
;       none.
;
;   USES.
;       all.

        ASSUME  CS:CGROUP, DS:NOTHING, ES:NOTHING, SS:NOTHING
DefProc Main, PUBLIC, FAR
```

```
        mov     ax, DGROUP
        mov     ds, ax
        ASSUME  DS:DGROUP               ; (DS) = DGROUP.

        mov     ax, MAXCHARS
        PRINTF  <This sample program has a main thread that allocates two\n>
        PRINTF  <worker threads.  Each thread prints its thread number in\n>
        PRINTF  <a loop $u times to show how preemption works in a\n>, <ax>
        PRINTF  <multithreaded system without prioritization.\n\n>

;       Allocate an event to trigger the worker threads to start running.

        mov     dl, SYS_ALLOCATE_EVENT  ; (DL) = function code.
        int     SYSINT                  ; (AX) = event handle.
        LJC     Main_Cleanup            ; if we couldn't allocate an event.
        PRINTF  <Trigger event has handle $u.\n>, <ax>
        mov     Trigger, ax             ; save trigger event handle.

;       Allocate an event for thread #1.

        mov     dl, SYS_ALLOCATE_EVENT  ; (DL) = function code.
        int     SYSINT                  ; (AX) = event handle.
        LJC     Main_Cleanup            ; if we couldn't allocate an event.
        PRINTF  <Event #1 has handle $u.\n>, <ax>
        mov     Done1, ax               ; save thread 1 "done" event handle.

;       Allocate an event for thread #2.

        mov     dl, SYS_ALLOCATE_EVENT  ; (DL) = function code.
        int     SYSINT                  ; (AX) = event handle.
        jc      Main_Cleanup            ; if we couldn't allocate an event.
        PRINTF  <Event #2 has handle $u.\n>, <ax>
        mov     Done2, ax               ; save thread 2 "done" event handle.

;       Call the AllocateThread kernel function for each thread we want
;       to create in the system (remember, we are running as a thread here
;       as well, so we will have two worker threads, plus this one makes
;       three threads in the system.  The function code is passed through
;       the DL register.  Other operands are passed in other registers
;       according to the specifications in the Technical Reference Manual.

        mov     ax, OFFSET CGROUP:Thread1Func
        mov     cx, CGROUP              ; (CX:AX) = FWA, thread function.
        mov     dl, SYS_ALLOCATE_THREAD ; (DL) = function code.
        int     SYSINT                  ; (AX) = new thread handle.
        jc      Main_Cleanup            ; if we couldn't allocate thread #1.
        PRINTF  <Thread #1 has handle $u.\n>, <ax>

        mov     ax, OFFSET CGROUP:Thread2Func
        mov     cx, CGROUP              ; (CX:AX) = FWA, thread function.
        mov     dl, SYS_ALLOCATE_THREAD ; (DL) = function code.
        int     SYSINT                  ; (AX) = new thread handle.
        jc      Main_Cleanup            ; if we couldn't allocate thread #2.
        PRINTF  <Thread #2 has handle $u.\n>, <ax>
```

```
;       Now set the trigger event to start the threads going; they are
;       currently waiting on our Trigger event.

        mov     ax, Trigger             ; (AX) = trigger event handle.
        mov     dl, SYS_SET_EVENT       ; (DL) = function code.
        int     SYSINT                  ; set the event.
        jc      Main_Cleanup            ; if we couldn't trigger the threads.

;       Now wait for both threads to complete their work.

        mov     ax, Done1               ; (AX) = event handle to wait on.
        mov     dl, SYS_WAIT_EVENT      ; (DL) = function code.
        int     SYSINT                  ; wait on the event.

        mov     ax, Done2               ; (AX) = event handle to wait on.
        mov     dl, SYS_WAIT_EVENT      ; (DL) = function code.
        int     SYSINT                  ; wait on the event.

;       Clean-up our objects here.  Both worker threads have deallocated
;       themselves because it was easiest to have them do it since they
;       have nothing better to do instead except spin and wait for us
;       to deallocate them with the provided handles.  Here we just return
;       the trigger and done events to the system.

Main_Cleanup:
        mov     ax, Trigger             ; (AX) = handle to deallocate.
        cmp     ax, INVALID             ; is the handle invalid?
        je      @f                      ; if so, skip this deallocate.
        mov     dl, SYS_DEALLOCATE_EVENT; (DL) = function code.
        int     SYSINT                  ; deallocate the event.

@@:     mov     ax, Done1               ; (AX) = handle to deallocate.
        cmp     ax, INVALID             ; is the handle invalid?
        je      @f                      ; if so, skip this deallocate.
        mov     dl, SYS_DEALLOCATE_EVENT; (DL) = function code.
        int     SYSINT                  ; deallocate the event.

@@:     mov     ax, Done2               ; (AX) = handle to deallocate.
        cmp     ax, INVALID             ; is the handle invalid?
        je      @f                      ; if so, skip this deallocate.
        mov     dl, SYS_DEALLOCATE_EVENT; (DL) = function code.
        int     SYSINT                  ; deallocate the event.

;       In all but error cases, our threads will have automatically
;       deallocated themselves, setting their handles to INVALID so
;       this code won't exit.  This is a nice error cleanup, though.

@@:     mov     ax, Thread1             ; (AX) = handle to deallocate.
        cmp     ax, INVALID             ; is the handle invalid?
        je      @f                      ; if so, skip this deallocate.
        mov     dl, SYS_DEALLOCATE_THREAD ; (DL) = function code.
        int     SYSINT                  ; deallocate the thread.
```

```
@@:     mov     ax, Thread2             ; (AX) = handle to deallocate.
        cmp     ax, INVALID             ; is the handle invalid?
        je      @f                      ; if so, skip this deallocate.
        mov     dl, SYS_DEALLOCATE_THREAD ; (DL) = function code.
        int     SYSINT                  ; deallocate the thread.
@@:

;       When we get here, both threads have completed their work.
;       Exit back to the DOS prompt with a DOSEXIT INT 21h call.

        PRINTF  <\nExample is finished.\n>

        mov     ah, DOSEXIT
        mov     al, 0                   ; successful status code.
        int     21h                     ; terminate program.
EndProc Main

UCODE   ENDS
        END     Main
```

Prioritized Scheduling

So far, we've focused on a very simple multithreaded program that shows how threads are created and destroyed. If you're going to be using the threaded model to design real-time applications, you will need more control over the scheduler so that time-critical tasks get precedence over background-type tasks. This is accomplished with prioritized scheduling, and Embedded DOS supports it directly.

Most embedded systems can benefit from prioritized threads. The two major categories of industrial embedded systems, control systems and data acquisition systems, need to interface with external equipment and interact with it quickly. At the same time, they must respond to operator input and refresh displays in the background. We'd like to assign a thread to each one of these jobs, but can't afford to let a low-priority job continue to run when a high-priority one must execute. Let's examine a simple data acquisition program that has four tasks:

1. The data collection routine that is executed when data is available.
2. The routine that writes buffered data to disk.
3. The screen updating routine.
4. The operator keyboard input routine.

To focus on the multithreading issues here and avoid getting bogged down in the details of reading data from a device, we will assume the existence of a few external routines that communicate with the hardware we're monitoring data from. First, we will need a routine called *StartDevice*, which will arm the device and enable delivery of interrupts to a routine that we specify as an argument. Naturally, there will be behind-the-scenes assembly-language "glue" that handles register saving, DS register initialization, etc. Second, we will assume the presence of *StopDevice*, a routine which disengages the asynchronous delivery of interrupts to our routine. Third, we will need a routine called *ReadDevice*, which reads a 16-bit value from the device it as its value.

Listing 5.3 shows the source for the multithreaded code that prioritizes our tasks. The data collection routine is entered at interrupt time, the highest priority possible. Next, we'll need a

high priority thread to write data to disk. We want the recording to happen instantly, even if the display updates more slowly. Finally, we have the screen updating task that displays the most-current value read from the device on the screen, and another task at the same priority level that uses our keyboard polling technique using *PassTimeSlice*.

```
#include <ktypes.h>
#include <kernel.h>
#include <system.h>
#include <conio.h>
#include <stdio.h>
#define HI_PRIORITY     (THREAD_PRIORITY_DEFAULT+1)
#define LO_PRIORITY     (THREAD_PRIORITY_DEFAULT)

extern VOID StartDevice (VOID (*InterruptRoutine)());
extern VOID StopDevice ();
extern USHORT ReadDevice ();

static USHORT StopThreads=FALSE;
static HANDLE Event1, Event2, DataReadyEvent;
static USHORT DataBuffer;

THREAD _loadds FlushDataToDisk ()
{
    FILE *DiskFile;
    USHORT BytesWritten;
    DiskFile = fopen ("myfile.dat", "w+b");
    while (!StopThreads) {
        WaitEvent (DataReadyEvent); // wait for more data.
        fwrite (DiskFile, DataBuffer, 2, &BytesWritten);
    }

    fclose (DiskFile);
    printf ("Data flushed to disk; data file closed.\n");
    SetEvent (Event1);
    DeallocateThread ();            // deallocate thread.
} // FlushDataToDisk

THREAD _loadds UpdateDisplay ()
{
    while (!StopThreads) {
        WaitEvent (DataReadyEvent); // wait for more data.
        printf ("\r%05u", DataBuffer);
    }
    SetEvent (Event2);
    DeallocateThread ();            // deallocate thread.
} // UpdateDisplay

interrupt _loadds DataReadyIsr ()
{
    DataBuffer = ReadDevice ();     // read data from device.
    PulseEvent (DataReadyEvent);    // start all threads processing.
} // DataReadyIsr

VOID main ()
{
    HANDLE Thread1, Thread2;
    UCHAR ch;

    AllocateEvent (&Event1);
    AllocateEvent (&Event2);
    Thread1 = AllocateThreadLong (FlushDataToDisk, 0, HI_PRIORITY);
    Thread2 = AllocateThreadLong (UpdateDisplay, 0, LO_PRIORITY);
```

```
    StartDevice (DataReadyIsr);     // allow data-ready interrupts.

    //
    // Since the root thread always executes at the default
    // priority (THREAD_PRIORITY_DEFAULT), and since we have
    // equated this application's LO_PRIORITY to the default,
    // we will be thinking of the root thread as a low priority
    // task in the system.  It simply round-robins with the
    // screen-updating thread, making sure the operator doesn't
    // want to terminate the application by pressing a key.
    //

    while (!kbhit ()) {       // check for a keystroke present.
        PassTimeSlice ();           // yield to worker threads.
    }
    ch = getch ();                  // accept the character.

    StopDevice ();                  // stop interrupts to DataReadyIsr.

    StopThreads = TRUE;
    WaitEvent (Event1);       // wait for first thread to exit.
    WaitEvent (Event2);       // wait for second thread to exit.
    DeallocateEvent (Event1); // release event to system.
    DeallocateEvent (Event2); // release event to system.
    printf ("Other threads have terminated themselves.\n");
} // main
```

By now you should be pretty comfortable with the idea of threads executing on their own, and using event objects to provide synchronization for startup and shutdown of your application. The program listed above uses an event for its main purpose-- to turn threads loose when something happens. In this case, it's when the *DataReadyIsr* routine gets control at interrupt time, reads a 16-bit datum from the hardware, and then pulses the event.

Of course, our example program has limitations. In real life, we will want the screen-updating thread to be fancier. It wouldn't use *printf*; it would probably use a windowing package to paint the screen with a number, or perhaps a bargraph. We'd also spend some time to think about how many data arrive each second, because if they arrive quickly, it would be far better to block the data into a bigger buffer, and write out the data in one big chunk. This will save quite a bit of time in the high priority thread.

Some performance tips are worth noting here. In our example, if you block-up data by even doubling the data buffer size, you will cut the CPU utilization (and request latency) by roughly 50%. This is tremendously significant, because it says that you can eliminate half of the disk writes in this program by simply declaring two USHORTs instead of one to hold data. *DataReadyIsr* needs to be marginally more intelligent to handle this, but it is probably worth it. The fully-optimal solution in very high performance data acquisition systems is to use a ring buffer for data, and have the *DataReadyIsr* routine trigger the event only when a sufficient amount of contiguous data is available in the ring buffer to make it worthwhile to flush it. If you are recording lots of data for each interrupt, consider using a ring buffer of pointers to data vectors stored in records. You can, of course, allocate this memory using *malloc*, or any other memory allocator (be careful not to allocate lots of kernel pool, however).

Using Timers for Synchronous Execution

Our data acquisition program depends on an external source of control (a hardware interrupt) to trigger the acquisition process that led to the capture and display of the data. When data must be

sampled on a regular basis as time-series data, consider using the Embedded DOS timer kernel object to provide a regular heartbeat that is tied to the PC hardware's 8253/8254 timer chip. Note: Embedded DOS times everything in milliseconds, not clock ticks. The OEM can use the Adaptation Kit to change the default 18.2 Hz tick rate (55ms) to something arbitrarily fast for improved granularity.

Let's write a similar program (below) that uses a timer object to repeatedly poll a device at regular intervals to drive the data collection process. Suprisingly, most of the code from the previous example stays the same.

```
#include <ktypes.h>
#include <kernel.h>
#include <system.h>
#include <conio.h>
#include <stdio.h>

#define HI_PRIORITY     (THREAD_PRIORITY_DEFAULT+1)
#define LO_PRIORITY     (THREAD_PRIORITY_DEFAULT)
#define TIMER_INTERVAL  100   // 100ms collection interval.

extern USHORT ReadDevice ();

static USHORT StopThreads=FALSE;
static HANDLE Event1, Event2, DataReadyEvent;
static HANDLE TimerHandle;
static USHORT DataBuffer;

THREAD _loadds FlushDataToDisk ()
{
    FILE *DiskFile;
    USHORT BytesWritten;
    DiskFile = fopen ("myfile.dat", "w+b");
    while (!StopThreads) {
        WaitEvent (DataReadyEvent); // wait for more data.
        fwrite (DiskFile, DataBuffer, 2, &BytesWritten);
    }

    fclose (DiskFile);
    printf ("Data flushed to disk; data file closed.\n");
    SetEvent (Event1);
    DeallocateThread ();            // deallocate thread.
} // FlushDataToDisk

THREAD _loadds UpdateDisplay ()
{
    while (!StopThreads) {
        WaitEvent (DataReadyEvent); // wait for more data.
        printf ("\r%05u", DataBuffer);
    }
    SetEvent (Event);
    DeallocateThread ();            // deallocate thread.
} // UpdateDisplay

void _loadds DataReadyTimerRtn (USHORT Dummy)
{
    StartTimer (TimerHandle, TIMER_INTERVAL);
    DataBuffer = ReadDevice ();     // read data from device.
    PulseEvent (DataReadyEvent);    // start all threads processing.
} // DataReadyTimerRtn

VOID main ()
{
```

```
    HANDLE Thread1, Thread2;
    UCHAR ch;

    AllocateEvent (&Event1);
    AllocateEvent (&Event2);
    Thread1 = AllocateThreadLong (FlushDataToDisk, 0, HI_PRIORITY);
    Thread2 = AllocateThreadLong (UpdateDisplay, 0, LO_PRIORITY);
    AllocateTimer (&TimerHandle, 0, DataReadyTimerRtn);
    StartTimer (TimerHandle, TIMER_INTERVAL);

    //
    // Since the root thread always executes at the default
    // priority (THREAD_PRIORITY_DEFAULT), and since we have
    // equated this application's LO_PRIORITY to the default,
    // we will be thinking of the root thread as a low priority
    // task in the system.  It simply round-robins with the
    // screen-updating thread, making sure the operator doesn't
    // want to terminate the application by pressing a key.
    //

    while (!kbhit ()) {             // check for a keystroke present.
        PassTimeSlice ();           // yield to worker threads.
    }
    ch = getch ();                  // accept the character.

    StopThreads = TRUE;
    StopTimer (TimerHandle);        // stop interrupts to DataReadyIsr.
    WaitEvent (Event1);             // wait for first thread to exit.
    WaitEvent (Event2);             // wait for second thread to exit.
    DeallocateEvent (Event1);       // release event to system.
    DeallocateEvent (Event2);       // release event to system.
    DeallocateTimer (TimerHandle);  // done with the timer.
    printf ("Other threads have terminated themselves.\n");
} // main
```

This quick tour of threads, events, and timers should have given you the flavor of programming real-time systems with Embedded DOS kernel objects. In the sections that follow, we will deal with the practical details that need to be addressed when writing larger systems.

Preemptive and Nonpreemptive Scheduling

Multitasking schedulers usually fall into two categories-- preemptive and non-preemptive. Preemptive schedulers allow tasks of the same priority to round-robin, usually switching context on a hardware timer interrupt, or sometimes even on every system call. Preemptive schedulers can be found in operating systems such as VAX/VMS, and OS/2.

Non-preemptive schedulers do exactly the opposite. A timer tick or system service request is no cause to give up the CPU to the next guy; instead, each task does as much work as it thinks "reasonable", and then calls a kernel function like Embedded DOS's *PassTimeSlice* function to yield the CPU to the next schedulable task. Non-preemptive schedulers can be found in operating systems such as Microsoft Windows and NetWare 386.

Both systems have their advantages. Preemptive systems guarantee that each thread within a priority level gets a chance to execute, even if one thread decides to go compute-bound and forgets to yield. When used wisely, these systems can have a more even response time to the external environment.

Non-preemptive systems, however, don't incur the overhead associated with constant context switching on every timer interrupt. In some systems, there is only one high priority thread in the system anyway, and switching back to the same thread that just got interrupted is a wasteful thing. Proponents of non-preemptive scheduling view their system as leaner, less wasteful of the CPU, and more deterministic.

Depending on your application, you will want to choose between these approaches. Telling Embedded DOS to select between these scheduling methods is trivial. It is done with the Embedded DOS critical section. By entering a critical section when your embedded program first gets control, and by leaving the critical section before it exits, timeslicing only occurs when threads voluntarily block on events, wait on mutex objects, or when they pass their timeslice. The next section will discuss us how to use critical sections.

System-Wide Critical Sections

There are two Embedded DOS kernel functions to manage the system-wide critical section count, which can nest up to 65535 times. When your program is first run, preemption is enabled, allowing context switches to occur at regular intervals due to the hardware timer interrupt from the 8253/8254 timer. The system critical section count is then 0.

To disable preemption, the *EnterCriticalSection* kernel function is called. This actually just increments the critical section count, and returns immediately, without context switching. Once this call returns, the calling thread has complete control over the system, so long as it does not cause a higher priority thread to become schedulable. This can happen because it creates one, or because it raises the priority of a lower-priority thread, or because it pulses/sets an event or releases a mutex object that a higher priority thread is waiting on. Ultimately, of course, all of these actions are voluntary on the part of the thread that caused non-preemption.

Once the system goes non-preemptive, more *EnterCriticalSection* calls simply increment the system critical section count. Corresponding calls to *LeaveCriticalSection* decrement the system critical section count.

When the last *LeaveCriticalSection* is called, causing the system critical section count to 0, the kernel forces a context switch, enabling other threads to quickly get control, because the scheduler assumes that the critical section may have lasted a considerable amount of time.

Guarding Critical Code/Data with Mutexes

While event objects are a sort of broadcasting mechanism that permits an ISR or thread to signal other threads waiting on the event, a mutex provides another form of synchronization.

Mutexes could easily be manufactured from event objects themselves, but could result in non-deterministic waiting if they were. Instead, they are built directly into the kernel, and provide a high-performance solution to one simple problem-- mutual exclusion.

Consider for example, a program (below) in which a non-reentrant function must be called by multiple threads. The function might be non-reentrant because it edits a data structure such as a linked list, or because it interacts with a piece of non-reentrant code in a C library, or perhaps because it interacts with a numeric coprocessor that doesn't know about threads. For whatever reason, we can use mutexes to make non-reentrant code reentrant at the thread level. The code is

still, of course, non-reentrant at the hardware interrupt level, but even this problem can be solved, as we will see in the section on ISRs.

Consider a system of two threads, one a supplier of manufactured objects that get stored at the end of a linked list, the other a consumer of these objects. The objects are simple data structures that get allocated and deallocated with *malloc* and *free*. This example doesn't serve any useful function on its own, but will illustrate how the mutex object can guard critical sections of code that manipulate fragile structures.

```
#include <ktypes.h>
#include <kernel.h>
#include <system.h>
#include <conio.h>

#define HI_PRIORITY     (THREAD_PRIORITY_DEFAULT+1)
#define LO_PRIORITY     (THREAD_PRIORITY_DEFAULT)

struct Object {
    struct Object *NextObject;
    USHORT DataItem;
}; // Object

static USHORT StopThreads=FALSE;
static HANDLE Event1, Event2;
static HANDLE ListMutex;            // mutex governing list, below.
static struct Object *ListHead=NULL; // linked list protected by mutex.

THREAD _loadds Consumer ()
{
    struct Object *p;

    while (!StopThreads) {
        AcquireMutex (ListMutex);   // guard linked list edit.
        if (ListHead != NULL) {
            p = ListHead;
            ListHead = p->NextObject;
            printf ("Received %u.\n", p->DataItem);
            free (p);
        }
        ReleaseMutex (ListMutex);   // done with linked list edit.
    }

    SetEvent (Event1);
    DeallocateThread ();            // deallocate thread.
} // Consumer

THREAD _loadds Supplier ()
{
    struct Object *p, *q;

    while (!StopThreads) {
        AcquireMutex (ListMutex);   // guard linked list edit.
        p = malloc (sizeof (struct Object));

        if (p != NULL) {
            if (ListHead == NULL) {
                ListHead = p;
                p->NextObject = NULL;
            } else {
                for (q=ListHead; q->NextObject != NULL; q=q->NextObject) ;
                q->NextObject = p;
                p->NextObject = NULL;
            }
            p->DataItem = DataCounter++
        }

        ReleaseMutex (ListMutex);   // done editing linked list.
    }

    SetEvent (Event2);
    DeallocateThread ();            // deallocate thread.
} // Supplier
```

```
VOID main ()
{
    HANDLE Thread1, Thread2;
    UCHAR ch;

    AllocateEvent (&Event1);
    AllocateEvent (&Event2);
    AllocateMutex (&ListMutex);
    Thread1 = AllocateThreadLong (Consumer, 0, LO_PRIORITY);
    Thread2 = AllocateThreadLong (Supplier, 0, LO_PRIORITY);

    while (!kbhit ()) {       // check for a keystroke present.
        PassTimeSlice ();           // yield to worker threads.
    }
    ch = getch ();                  // accept the character.

    StopThreads = TRUE;
    WaitEvent (Event1);             // wait for first thread to exit.
    WaitEvent (Event2);             // wait for second thread to exit.
    DeallocateEvent (Event1);       // release event to system.
    DeallocateEvent (Event2);       // release event to system.
    DeallocateMutex (ListMutex);    // done with the mutex.
    printf ("Other threads have terminated themselves.\n");
} // main
```

Interrupt Service Routines

At this point you should have a good understanding of task-time execution in Embedded DOS. You can use threads, timers, events, and mutexes to synchronize your own tasks, and prioritize your system. Now it's time to take a look at some background material that will be helpful when you need to write interrupt service routines that must interface with your C code.

There are a few basic points that have to be addressed in every ISR you will write, since they must be perfect or your system won't last past the next timer tick, if that long. The next example shows a simple ISR that is hooked on a hardware interrupt. The "hooking" of this interrupt is done in a little routine called *HookIsr*, called by the C main routine.

This ISR implements only the basics; the minimum necessary to get a C-language routine going from inside interrupt context. When the interrupt is invoked, the interrupted task's flags and 16:16 return address are saved on the stack. The ISR pushes all of the other general registers, initializes DS=DGROUP (not required for all memory models, but typical for large-model MS-C programs). The direction flag is cleared with a CLD instruction, so that C library routines like *strcmp* scan in the forward direction. Then the C interrupt handler is called as a far procedure. It executes, and returns control to the ISR, which restores the general registers and returns into the interrupted thread's context.

The conspicuous "MOV AL, 20h" and "OUT 20h, AL" instructions re-arm the primary 8259 interrupt controller, allowing interrupts to arrive from the device again. Typically, the device also has to be re-armed here, too. You should re-arm the 8259 before you call your C routine if you will be executing for a long time. On the other hand, you should re-arm the 8259 after you call your C routine if you need to access your device's hardware registers before you dismiss the hardware interrupt.

```
        EXTRN   _DataReadyIsr:FAR   ; FAR routine in C module.

CODE    SEGMENT PARA PUBLIC 'CODE'
```

```
CODE    ENDS
CGROUP  GROUP   CODE, ... other code segments ...
DATA    SEGMENT PARA PUBLIC 'DATA'
DATA    ENDS
DGROUP  GROUP   DATA, ... other data segments ...

CODE    SEGMENT PARA PUBLIC 'CODE'
        ASSUME  CS:CGROUP, DS:NOTHING, ES:NOTHING, SS:NOTHING

Isr     PROC    FAR                 ; called via interrupt.
        push    ax                  ; save all registers.
        push    bx
        push    cx
        push    dx
        push    si
        push    di
        push    bp
        push    ds
        push    es
        mov     al, 20h
        out     20h, al         ; reenable interrupts at 8259.
        mov     ax, DGROUP
        mov     ds, ax              ; (DS) = DGROUP.
        ASSUME  ds:DGROUP
        cld                         ; forward string copies.
        call    _DataReadyIsr       ; execute C code.
        pop     es                  ; restore all registers.
        pop     ds
        pop     bp
        pop     di
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        iret                        ; return from interrupt.
Isr     ENDP

        PUBLIC  _HookIsr            ; allow access from C.
_HookIsr PROC   FAR
        sub     ax, ax
        mov     es, ax              ; (ES) = seg FWA, int. vector table.
        mov     word ptr es:[0ch*4+0], OFFSET CGROUP:Isr
        mov     word ptr es:[0ch*4+2], CGROUP
        ret                         ; return to caller.
_HookIsr ENDP

CODE    ENDS
        END
```

There are some things you should never attempt to do inside an ISR. As far as the kernel is concerned, your ISR should never ask for a resource that could result in a deadlock. Doing so would hang the machine. Consider what happens if one of your threads has just acquired a mutex governing a structure it desires to edit. Right in the middle of that structure manipulation, a hardware interrupt executes the ISR. Your ISR breaks into the thread, and attempts to acquire the same mutex so that it can edit the structure itself. Which thread context are we executing in? The same thread that has already acquired the mutex. The ISR will block indefinitely, waiting for our interrupted thread to release the mutex.

There is a rule to follow here to fix the problem. If critical code or resources must be shared between threads and interrupt code, then you must either disable interrupts during the critical

section, or you must use task-time mutual exclusion, and have the ISR cause a thread to be executed to do its work.

A very attractive way to accomplish this is to simply call the *AllocateThread* function right in your ISR, so that all of its work gets done at task time in the thread function you select. When the thread finishes, it can simply destroy itself. This method has the advantage that a newly-allocated thread executes on its own fresh stack. Typically, DOS programs have little stack available for ISRs to execute on. The next listing shows an assembly module that performs substantially the same function as the previous ISR, except that it allows the C routine to do its work in the context of a separate worker thread.

```
        EXTRN   _DataReadyIsr:FAR   ; FAR routine in C module.
        INCLUDE DEFINES.INC         ; kernel definitions.
        INCLUDE MACROS.INC          ; kernel macros.

CODE    SEGMENT PARA PUBLIC 'CODE'
CODE    ENDS
CGROUP  GROUP   CODE, ... other code segments ...

DATA    SEGMENT PARA PUBLIC 'DATA'
DATA    ENDS
DGROUP  GROUP   DATA, ... other data segments ...

CODE    SEGMENT PARA PUBLIC 'CODE'
        ASSUME  CS:CGROUP, DS:NOTHING, ES:NOTHING, SS:NOTHING

;       The following routine is executed by a worker thread
;       allocated inside the ISR.  Once this code begins running,
;       the interrupted thread may resume executing, causing
;       the ISR to return.  You should not expect either to
;       necessarily finish before the other.

TaskIsr PROC    FAR
        mov     ax, DGROUP
        mov     ds, ax              ; (DS) = DGROUP.
        ASSUME  ds:DGROUP
        cld                         ; forward string copies.
        call    _DataReadyIsr       ; execute C code.
        DeallocateThread            ; macro to deallocate thread.
TaskIsr ENDP

Isr     PROC    FAR                 ; called via interrupt.
        push    ax                  ; save all registers.
        push    bx
        push    cx
        push    dx
        push    si
        push    di
        push    bp
        push    ds
        push    es
        mov     al, 20h
        out     20h, al             ; reenable interrupts at 8259.
        mov     cx, cs
        mov     ax, OFFSET CGROUP:TaskIsr ; (CX:AX) = task time routine.
        mov     dl, SYS_ALLOCATE_THREAD   ; (DL) = kernel function number.
        int     2dh                 ; run the thread in parallel.
        pop     es                  ; restore all registers.
        pop     ds
        pop     bp
        pop     di
```

```
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        iret                        ; return from interrupt.
Isr     ENDP
CODE    ENDS
        END
```

The ISR above uses the assembly language equivalent of the *AllocateThread* macro. You should pay careful attention to your priority scheme, if one exists, so that the task-time ISR routine executes at the appropriate priority level. If necessary, you may need to use the **SYS_ALLOCATE_THREAD_LONG** kernel function (see the kernel API chapter for a discussion of this) to allocate a thread with a priority other than the system default priority, **THREAD_PRIORITY_DEFAULT**.

Using Your C Compiler and Runtime Library

The simple programs we've studied in this chapter rely on an assumption that the C library functions can execute from different thread contexts, and that the compiler generates correct code to run in a multitasking environment.

In practice, there are several issues that must be addressed in a production program to produce good results. Depending on your compiler switches, which library functions you call, and other variables, you may have difficulty in getting some of your code that uses Embedded DOS kernel or executive services to work correctly. This section will enumerate the most important issues that you must address in multitasking programs running under Embedded DOS.

Assumptions About SS and DS

When a program is run under a single-tasking DOS, control first transfers to the C runtime library's initialization routine, which initializes its own variables and sets up a stack and heap immediately following the program's load image. Then, it calls DOS to reduce the size of memory allocated for the program to just what it needs plus the stack. All vendors' C compilers do this, and the runtime library routines can sometimes make assumptions about the values stored in SS (normally, a program never alters this register) and DS (sometimes, a program alters this register).

The first assumption in many programs is that SS=DGROUP. This common optimization is used where a small amount of program data can be located at the base of the heap, allowing the stack to grow down towards it. Using only one segment register (SS) to address this data, this optimization allows the runtime routines to assume that, even though they have DS=DGROUP, they can use SS to refer to global data.

The problem with this assumption is that, in a multitasking environment, each thread needs to run on its own stack. The program initially runs in the context of a single thread (with a system stack), and the root thread executes the C runtime library's init code, which switches stacks to its own private stack. Thus, the code executed in the context of the root thread can make the SS=DGROUP assumption, but not the other threads in the system. To solve this problem, you'll need to disable this optimization with a compile-time switch or pragma.

A problem related to the SS=DGROUP assumption is another assumption commonly made by some compilers (MSC): SS=DS. In some cases, a compiler will generate code that assumes that the DS register is always loaded with a pointer to the data segment, which is the same as the stack segment. This makes it impossible for allocated threads to refer to global data. Nearly all compilers have a compile-time switch (see your language vendor's user's guide, or contact their technical support) to turn this assumption off.

Finally, threads enter the system with arbitrary values stored in their registers (except for CS:IP and SS:SP, of course). This means DS and ES are not initialized. Some compilers assume that, upon entry to a function, DS=DGROUP. If yours does, then the new threads you allocate will not be able to access global data until you set DS properly.

In compilers where this is a problem, you can initialize DS properly by using the _loadds directive shown below at the beginning of a thread function (it only needs to be done by each thread when it begins execution). Note: Most compilers have a switch to disable this assumption as well.

```
THREAD _loadds ThreadFunc ()
{
     ...
     ... continue with thread body ...
     ...
} // ThreadFunc
```

Stack Probes

Many compilers implement a feature called stack probes that inserts calls to a stack checking routine at the beginning of every compiled function. The stack checking routine, located in the C runtime library, determines if the stack segment or pointer have exceeded their bounds, and if so, the routine prints the following well-known message and halts the program:

```
R6000: STACK OVERFLOW
```

While meant as a way to detect stack overflow problems that can only be detected dynamically, this checking of SS and SP gives multitasking programs problems, because the assignment of SS and SP is up to the operating system, not the C library.

Most compilers have a compile-time switch to eliminate stack probes in the compiled code. Unfortunately, most C compiler vendors pride themselves in having compiled the C runtime library with their own compiler, usually with stack probes enabled. This is a fatal flaw in the runtime library. If your vendor's C runtime library has this flaw, it means you can't use *printf*, *malloc*, and a host of other important functions in your threads; they can only be used in the main routine.

We do have ways around this problem. The best way is to purchase the source code with your compiler (Borland supplies this) and recompile without stack probes enabled. Second best is to purchase a third party library with source that avoids the problem.

The final solution to this problem is to write the *main* function as a server thread, which receives requests to perform work on the C library through message ports or queues.

Runtime Library Reentrancy

Assuming all of the addressability and stack probe issues have been resolved, it is possible that you may have difficulty in executing C library functions from several threads at the same time.

The problem here is that, depending on the compiler vendor's implementation, the functions may use static or global variables to maintain their state when processing some function (*printf* is notorious for this problem.) Two simultaneous calls to the same function or functions in the same family could cause the functions to crash.

To solve this problem, the application code must be structured to call these functions in a serialized fashion. You can serialize the calls yourself, or you can use the Embedded DOS **mutex** object to guard the calls as critical resources.

Finally, it is possible to write wrapper routines that serialize access with the mutex object so that mutex-based code is not littered throughout your code. The next code fragment illustrates how the client/server model can be used to implement an fopen call that serializes access to the real C library *fopen* routine. Naturally, the new *fopen* routine's name has been altered slightly to not conflict with the C library's routine.

```
HANDLE PrivateMutex;

FILE *_fopen (char *path, char *type)
{
      FILE *f;
      AcquireMutex (PrivateMutex);
      f = fopen (path, type);
      ReleaseMutex (PrivateMutex);
      return f;
} // _fopen

THREAD _loadds Thread1 ()
{
      FILE *myfile;
      myfile = _fopen ("myfile.dat", "r+b");
      ...
} // myfunc

void main ()
{
      FILE *mainfile;
      AllocateMutex (&PrivateMutex);
      AllocateThread (Thread1);
      ...
      mainfile = _fopen ("file2.dat", "w+b");
} // main
```

Using Embedded DOS With Modula-2 and Ada

While C has no tasking model in the language definition, some languages have a predefined tasking model. Modula-2 and Ada are the important commercial languages that provide this support in the runtime library, and they require extra consideration before these systems can run under Embedded DOS.

The actual language multitasking on top of Embedded DOS's multitasking is not a problem. Nearly all multitasking real-time kernels and executives run on top of Embedded DOS. There are two issues that must be addressed in order for both scheduling systems to work in tandem:

1. Embedded DOS supports reentrant file I/O by using mutexes to guard critical sections of code and data. If a thread blocks on a mutex in the file system, it will run again because another thread will release the mutex soon. However, allocating two Modula-2 or Ada tasks does not guarantee that these tasks will run as separate Embedded DOS threads. If they run in the same Embedded DOS thread context, then it is possible to make DOS calls in separate Modula-2 tasks that run in the same Embedded DOS thread, causing a thread to block itself. This causes deadlock.

The solution to this problem is to let Embedded DOS schedule tasks, and tie the language's scheduler to the Embedded DOS kernel's thread object, or use Embedded DOS threads alone, without the language tasking.

2. When Embedded DOS switches threads, the language's task does not know that it must switch tasks as well. To solve this problem, you can use the **SetThreadInformation** kernel function to register context switching appendages, allowing the language's scheduler to gain control right before Embedded DOS schedules a thread, and right before the thread blocks. See the chapter on the Kernel API for details.

Chapter 6

KERNEL API

This chapter describes the *kernel* of the Embedded DOS operating system. The purpose of the kernel is to provide a software abstraction of the underlying hardware through the implementation of lightweight kernel-level objects.

The kernel itself only implements these objects; it does not handle I/O, memory management (beyond implementation of the pool object), and does not handle program loading, or file system functions. These other system components are separate modules that call the kernel as clients of the kernel to use the elemental objects; they are considered a part of the *system*, but not a part of the kernel.

Objects supported by the kernel are as follows:

- THREAD — Basic source of asynchronous execution.
- TIMER — Basic source of synchronous execution.
- EVENT — Broadcast-type semaphore.
- MUTEX — Mutual exclusion semaphore.
- SPINLOCK — Multiprocessor synchronization semaphore.
- NAMED OBJECT — System-wide name space for user/system objects.
- POOL — Centralized interrupt-time memory manager.

The kernel implements each object by providing a set of services to allocate, deallocate, query, and manipulate each object. These services are provided through a Embedded DOS-proprietary API, utilizing software interrupt 2dh. An assembly library is available that marshals C-language arguments into registers and calls INT 2dh on behalf of its clients. This enables C-language applications to use kernel functions.

All kernel services are extremely fast, notwithstanding the specific interface used to deliver the request to the kernel from the system component or the application program. Kernel services may be requested at any time, unless explicitly stated otherwise in this specification. For example, it is perfectly acceptable to allocate system pool memory at interrupt time.

Thread Object

The basic unit of execution in the Embedded DOS operating system is the *thread*, and is implemented in the kernel with the thread object. Threads may be allocated and deallocated at any time, including interrupt time.

The kernel schedules the execution of a thread whenever a processor becomes free to execute a thread, and a thread is executable. Threads may voluntarily release control over the processor when they wait on events or mutexes. They also involuntarily release control of the processor when a timer interrupt occurs and the kernel decides to give another thread CPU cycles.

A thread's operating context consists simply of an initial stack segment with room to save the entire programmable register set of the CPU. For 8088, 8086, 80188, 80186, 80288, 80286, 80386, 80486, and Pentium processors, this includes the following registers:

AX, BX, CX, DX, SI, DI, DS, ES, BP, CS, IP, SS, SP, FL

Thirty-two bit registers in 80386, 80486, Pentium, and above processors are saved if the appropriate flag is raised with a call to the **SetThreadInformation** Kernel function. The state of the numeric coprocessor may be saved in user thread appendage routines (see **SetThreadInformation**).

One of the more typical uses of the thread object in the system is as a source of task-time control in an interrupt service routine (ISR). For example, the disk driver's BPB aging timer expiration routine receives control from the kernel at interrupt time, schedules a thread for execution to handle the timeout, and then returns control to the caller in parallel.

AllocateThread

A thread object is created with the **AllocateThread** kernel function. Once the thread is allocated, it is automatically and independently scheduled for execution by the kernel until it is deallocated or aborted.

Thread allocation is a lightweight operation. It is acceptable to create a thread at interrupt time to delay the bulk of the work to task time. The kernel immediately schedules the new thread for execution, so ISRs must first perform any interrupt cleanup operations before allocating the new thread. Failure to re-arm the 8259 before allocating a new thread in interrupt context could result in an interval of up to 55ms during which interrupts would be masked (this interval is based on a hardware timer timebase of 18.2 ticks/second for the typical PC implementation; the actual number may vary, depending on the OEM adaptation).

It is important for the thread function to load any segment registers it requires for sharing data with other threads in the system. The DS and ES registers are not necessarily the same as the parent thread; making it necessary to load the DS register upon entry to a function that will be using static variables addressed with DGROUP. You can cause your compiler to automatically generate the code to load DS with DGROUP with a compiler command line switch.

This short form thread allocator automatically allocates some kernel pool for the thread's stack, and initializes the thread's priority to **THREAD_PRIORITY_DEFAULT**, or 16384.

Upon return, the assembly language software interrupt clears the carry flag if the operation was successful, and sets it if the operation was not performed. The high-level language interface returns a non-zero handle if successful, else a zero value if unsuccessful.

Assembly Language Format:

```
mov   ax, OFFSET CGROUP:TargetLabel
mov   cx, CGROUP
mov   dl, SYS_ALLOCATE_THREAD
int   2dh                          ; (AX) = new thread handle.
jc    Failure
```

Macro Instruction Format:

label **AllocateThread** <*TargetLabel*>

Portable Request Format:

STATUS AllocateThread(
IN VOID far (*TargetLabel***)()**
);

Parameters:

TargetLabel - Specifies far address of the label or function that the new thread will begin executing at. For the Macro Instruction format only, the label is assumed to be relative to the parent's CS register. The C-language interface requires a FAR pointer to a function.

AllocateThreadLong

A higher degree of control over thread allocation is accomplished with the long form kernel function, **AllocateThreadLong**. This function allows the caller to specify the priority at which the thread is to start running, and an optional stack segment to be used for the thread. The stack segment must be at least 1Kb in size. Once the thread is allocated, it is automatically and independently scheduled for execution by the kernel until it is deallocated or aborted.

To pass parameters to a child thread, use the **AllocateThreadLong** function to create a low priority thread (say, priority 0x0000), and then use the **SetThreadInformation** function to change the P1 and P2 parameters for the thread. Upon restoration of the thread's priority with the **PrioritizeThread** function, the child thread will be able to read these values with the **QueryThreadInformation** function.

Upon return, the assembly language software interrupt clears the carry flag if the operation was successful, and sets it if the operation was not performed. The high-level language interface returns a non-zero handle if successful, else a zero value if unsuccessful.

Assembly Language Format:

```
mov   ax, OFFSET CGROUP:TargetLabel
mov   cx, CGROUP
```

```
mov     bx, <Priority>          ; (BX) = thread's initial priority.
mov     es, SEG Stack                   ; (ES) = stack segment to use.
mov     dl, SYS_ALLOCATE_THREAD_LONG
int     2dh                             ; (AX) = new thread handle.
jc      Failure
```

Macro Instruction Format:

none.

Portable Request Format:

STATUS AllocateThreadLong(
IN VOID far (*TargetLabel***)(),**
IN USHORT *StackSegment***,**
IN USHORT *Priority*
);

Parameters:

TargetLabel - Specifies the far address of the label or function at which the new thread will begin executing.

StackSegment - A 16-bit segment value that specifies the segment address of a 1K stack to be used by the thread. If the specified value is zero, then a stack will be automatically allocated from the kernel's memory pool, as is the case with **AllocateThread**.

Priority - A 16-bit value that specifies the initial scheduling priority for the new thread. Higher values have more urgent priority than lower values. By definition, The **AllocateThread** allocator creates threads with a priority equal to **THREAD_PRIORITY_DEFAULT** (16384). The following priority values are pre-architected:

THREAD_PRIORITY_LOW - Lowest possible priority.
THREAD_PRIORITY_DEFAULT - The default priority.
THREAD_PRIORITY_HI - The highest priority.
THREAD_PRIORITY_RESERVED - The lowest reserved priority.

DeallocateThread

A thread object is removed from the system with the **DeallocateThread** kernel function. Once the thread is deallocated, its object is returned to the system to be recycled, and another thread is scheduled for execution. If no other threads in the system are available for execution, then the scheduler executes the idle loop until an interrupt routine is executed which causes a new thread to be allocated or a blocked thread to be released.

A thread may only remove itself from the system with this function. A thread may remove another thread in the system through the **AbortThread** function, although this function must be used with care as it does not clean-up acquired resources.

Assembly Language Format:

```
mov   dl, SYS_DEALLOCATE_THREAD
int   2dh                       ; never returns (current thread dies).
```

Macro Instruction Format:

label **DeallocateThread**

Portable Request Format:

STATUS DeallocateThread();

Parameters:

none.

AbortThread

A thread may remove another thread object in the system with the **AbortThread** kernel function. The kernel does not perform cleanup with regard to resources held by the target thread, including but not limited to pool, events, mutexes, timers, or spinlocks.

A thread may remove itself from the system with this function by specifying a thread handle equal to zero. This form has identical results to calling the **DeallocateThread** function.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   dl, SYS_ABORT_THREAD
mov   ax, <ThreadHandle>   ; (AX) = handle of thread to abort.
int   2dh                       ; destroys target thread.
jc    Failure
```

Macro Instruction Format:

none.

Portable Request Format:

STATUS AbortThread(
IN HANDLE *ThreadHandle*
);

Parameters:

ThreadHandle - A 16-bit handle to a thread object as returned by the **AllocateThread** or **AllocateThreadLong** kernel functions. If the specified value is zero, then the current thread aborts itself.

PrioritizeThread

A thread may change its priority with the **PrioritizeThread** kernel function. If a thread lowers its priority, then the kernel automatically performs a rescheduling to allow other threads with potentially higher priorities to run.

Assembly Language Format:

```
mov   dl, SYS_PRIORITIZE_THREAD
mov   ax, <ThreadHandle>  ; (AX) = thread handle.
mov   cx, <Priority>      ; (CX) = new thread priority.
int   2dh                 ; change the priority.
jc    Failure
```

Macro Instruction Format:

none.

Portable Request Format:

STATUS PrioritizeThread(
IN HANDLE *ThreadHandle***,**
IN USHORT *Priority*
);

Parameters:

ThreadHandle - A 16-bit handle to a thread object as returned by the **AllocateThread** or **AllocateThreadLong** kernel functions. If the specified value is zero, then the current thread's priority is adjusted.

Priority - A 16-bit value specifying the new priority to assign the thread. The priority must be greater than or equal to **THREAD_PRIORITY_LOW** and less than **THREAD_PRIORITY_RESERVED** to be valid. Note that the operating system normally has a priority one (0x0001) thread in the system, and **THREAD_PRIORITY_LOW** is less than that, so threads assigned the lowest priority normally will never execute. The following priorities are predefined:

THREAD_PRIORITY_LOW - Value 0, the lowest priority in the system, executed only if no higher priority threads exist in the system.

THREAD_PRIORITY_DEFAULT - Value 16384, the standard priority assigned to threads when allocated with the **AllocateThread** kernel function.

THREAD_PRIORITY_HI - Value 32767, the highest valid priority in the system, preempting all other priorities except itself.

THREAD_PRIORITY_RESERVED - Value 32768, the lowest priority reserved for internal system use. Priorities greater than or equal to this priority are not for application use; they have special internal meaning to the system.

QueryThreadHandle

A thread function may obtain its own handle by calling the **QueryThreadHandle** kernel function. This allows thread functions to store the handle for later reference, or to pass the handle to other routines, or even to supply it to thread-manipulation functions to operate on itself (note that handle 0 can be used to do this as well).

This function can also be used at interrupt time to return the handle of the interrupted thread, so that interrupt routines can make limited task-time decisions.

Assembly Language Format:

```
mov   dl, SYS_QUERY_THREAD_HANDLE
int   2dh                  ; (AX) = thread handle.
```

Macro Instruction Format:

none.

Portable Request Format:

STATUS QueryThreadHandle(
OUT PHANDLE *ThreadHandle*
);

Parameters:

ThreadHandle - A 16:16 pointer to a 16-bit storage location where the handle to the current thread object is returned, as it was returned by the **AllocateThread** or **AllocateThreadLong** kernel functions.

QueryThreadInformation

The **QueryThreadInformation** kernel function offers a general interface for querying the attributes of an existing thread in the system. Most information set by the SetThreadInformation can be queried with this function. The user-defined kernel appendages (**THREAD_INFOTYPE_URUN, THREAD_INFOTYPE_USTOP, and THREAD_INFOTYPE_UEXIT**) are exceptions to this rule.

Assembly Language Format:

```
mov   dl, SYS_QUERY_THREAD_INFO
mov   ax, <ThreadHandle>   ; (AX) = thread handle.
mov   cx, <InfoType>               ; (CX) = information type ID.
```

```
les     bx, <Buffer>        ; (ES:BX) = info pointer.
int     2dh                         ; set the information.
jc      Failure
```

Macro Instruction Format:

none.

Portable Request Format:

STATUS QueryThreadInformation(
IN HANDLE *ThreadHandle***,**
IN USHORT *InfoType***,**
IN PVOID *Buffer*
);

Parameters:

ThreadHandle - A 16-bit handle to a thread object as returned by the **AllocateThread** or **AllocateThreadLong** kernel functions. If the specified value is zero, then the current thread is queried.

InfoType - A 16-bit unsigned value specifying the type of information to be queried for the specified thread. Only architected information types are supported. The following information types are defined:

THREAD_INFOTYPE_URUN - Not supported by **QueryThreadInformation**.

THREAD_INFOTYPE_USTOP - Not supported by **QueryThreadInformation**.

THREAD_INFOTYPE_UEXIT - Not supported by **QueryThreadInformation**.

THREAD_INFOTYPE_NAME - Specifies that thread's name will be returned in the buffer. The buffer must be large enough to accommodate the size of the name (a suggested size of 64 bytes is enough for most naming conventions; the kernel will by default select names smaller than this.)

THREAD_INFOTYPE_P1 - Specifies that the first 32-bit value passed to the thread will be returned in the specified buffer. The buffer must be 4 bytes long to accommodate this parameter, which may need to be casted to a longword or a pointer, as necessary, to convert back to the type of data used in the SetThreadInformation function.

THREAD_INFOTYPE_P2 - Specifies that the second 32-bit value passed to the thread will be returned in the specified buffer. The buffer must be 4 bytes long to accommodate this parameter, which may need to be casted to a longword or a pointer, as necessary, to convert back to the type of data used in the SetThreadInformation function.

Buffer - A 32-bit segment:offset pointer to a storage location interpreted according to the contents of the *InfoType* field. The size of this storage area is dictated by the information type.

SetThreadInformation

The **SetThreadInformation** kernel function offers a general interface for manipulating the attributes of an existing thread in the system. User appendages to the kernel, and special operating characteristics, can be set with this function.

Parameters may also be passed to child threads with **SetThreadInformation** by using the **THREAD_INFOTYPE_P1** and **THREAD_INFOTYPE_P2** information types. To use this feature, use **AllocateThreadLong** to create the new child thread, specifying a priority lower than the current thread, so that the child thread does not run until after the **SetThreadInformation** function can be used to install the parameters in the child thread. Then, call **SetThreadInformation** to pass the parameters, and afterward, raise the priority of the child with **PrioritizeThread**.

Assembly Language Format:

```
mov     dl, SYS_SET_THREAD_INFO
mov     ax, <ThreadHandle>  ; (AX) = thread handle.
mov     cx, <InfoType>              ; (CX) = information type ID.
les     bx, <Buffer>        ; (ES:BX) = info pointer.
int     2dh                         ; set the information.
jc      Failure
```

Macro Instruction Format:

none.

Portable Request Format:

STATUS SetThreadInformation(
IN HANDLE *ThreadHandle***,**
IN USHORT *InfoType***,**
IN PVOID *Buffer*
);

Parameters:

ThreadHandle - A 16-bit handle to a thread object as returned by the **AllocateThread** or **AllocateThreadLong** kernel functions. If the specified value is zero, then the current thread is affected.

InfoType - A 16-bit handle to a thread object as returned by the **AllocateThread** or **AllocateThreadLong** kernel functions. If the specified value is zero, then the current thread's priority is adjusted. The following information types are defined:

THREAD_INFOTYPE_URUN - Specifies that the buffer pointer points to a user-defined routine that is to be executed by the kernel's thread scheduler

whenever a thread is dispatched (immediately before the user code is given control). The user-defined routine must never assume the valid contents of any general registers, and may destroy all its general registers during execution. The user-defined routine is called with interrupts enabled, and must return with a RETF (far return) instruction.

THREAD_INFOTYPE_USTOP - Specifies that the buffer pointer points to a user-defined routine that is to be executed by the kernel's thread scheduler whenever a thread's context is saved (immediately after being blocked or when its time-slice has elapsed). The user-defined routine must never assume the valid contents of any general registers, and may destroy all its general registers during execution. The user-defined routine is called with interrupts enabled, and must return with a RETF (far return) instruction.

THREAD_INFOTYPE_UEXIT - Specifies that the buffer pointer points to a user-defined routine that is to be executed by the kernel's thread scheduler when the thread terminates itself with the **DeallocateThread** kernel function (this routine is not executed when the thread is aborted with the **AbortThread** kernel function.) The user-defined routine must never assume the valid contents of any general registers, and may destroy all its general registers during execution. The user-defined routine is called with interrupts enabled, and must return with a RETF (far return) instruction.

THREAD_INFOTYPE_FLAGS - Specifies that the buffer pointer points to a 16-bit word containing bitflags that, when set, indicate that the specified capabilities are enabled. The following are the defined capabilities:

THREAD_CAPABILITY_386 - The entire contents of the CPU's 32-bit registers (EAX for example) rather than just the low 16-bit halves of these registers, will be saved and restored as a part of the thread's context switch.

THREAD_CAPABILITY_X87 - The state of the numeric coprocessor is saved as a part of the thread's context switch. For future use only; do not use.

THREAD_INFOTYPE_NAME - Specifies that the buffer pointer points to an ASCIIZ string that names the thread. This name is displayed in the debugger's thread display.

THREAD_INFOTYPE_P1 - Specifies that the buffer pointer is to be interpreted as a 32-bit value to be passed to threads newly-allocated by this parent thread as its first 32-bit parameter.

THREAD_INFOTYPE_P2 - Specifies that the buffer pointer is to be interpreted as a 32-bit value to be passed to threads newly-allocated by this parent thread as its second 32-bit parameter.

Buffer - A 32-bit segment:offset pointer, interpreted according to the contents of the *InfoType* field. In some cases, this pointer is the data (for example, a pointer to a user-specified routine). In other cases, this pointer points to a user-defined buffer that contains information to be passed to the thread.

EnterCriticalSection

A thread may disable context switching with the **EnterCriticalSection** kernel function. The kernel allows nesting of **EnterCriticalSection** calls and counts them as the number of reasons why round-robin schedulings within a priority should not take place. The **LeaveCriticalSection** decrements this counter.

After the **EnterCriticalSection** function returns, context switching is disabled. This is useful when manipulating data or hardware that cannot be shared among tasks, but that is under both task-time control and interrupt-time control.

Assembly Language Format:

```
mov   dl, SYS_ENTER_CRITICAL_SECTION
int   2dh                          ; go non-preemptive.
```

Macro Instruction Format:

none.

Portable Request Format:

STATUS EnterCriticalSection();

Parameters:

none.

LeaveCriticalSection

A thread may reenable context switching with the **EnterCriticalSection** kernel function. The kernel allows nesting of **EnterCriticalSection** calls and counts them as the number of reasons why round-robin schedulings within a priority should not take place.

The **LeaveCriticalSection** decrements this counter, keeping context switching disabled until the **EnterCriticalSection** nesting level is zero. When this happens, a forced rescheduling occurs.

Assembly Language Format:

```
mov   dl, SYS_LEAVE_CRITICAL_SECTION
int   2dh                          ; allow preemption.
```

Macro Instruction Format:

none.

Portable Request Format:

STATUS LeaveCriticalSection();

Parameters:

none.

PassTimeSlice

A thread may cause a forced rescheduling to occur with the **PassTimeSlice** kernel function. Regardless of the critical section level, this function immediately saves the state of the current thread, and looks for the next highest priority thread in the system, which may be the same thread.

The reader should be advised that calling **PassTimeSlice** while the critical section level is non-zero will possibly result in another thread gaining control with preemption disabled. If the newly-executing thread is not prepared to decrement the critical section level on behalf of the first thread, or if it is not prepared to issue its own call to **PassTimeSlice**, the original thread might not again receive control.

Assembly Language Format:

```
mov   dl, SYS_PASS_TIME_SLICE
int   2dh                     ; force context switch.
```

Macro Instruction Format:

none.

Portable Request Format:

STATUS PassTimeSlice();

Parameters:

none.

Event Object

The *event* object is the basic mechanism used to control synchronous dispatching of threads. An event is assigned a state, and must always be in either the *set* or *cleared* states.

Facilities are provided by the kernel to set, clear, and pulse events. The kernel also provides allocation and deallocation services to make event objects. Threads may inspect the state of an event object, and may wait on an event until the event has reached the *set* state.

AllocateEvent

An event object is created with the **AllocateEvent** kernel function. Once the event is allocated, it is automatically maintained by the kernel in response to service requests.

This function returns a handle to an event object that must be used by the caller in subsequent kernel services that operate on events. The kernel initializes the event object to the *cleared* state.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   dl, SYS_ALLOCATE_EVENT
int   2dh                    ; (AX) = event handle.
jc    Failure
```

Macro Instruction Format:

label **AllocateEvent** *<EventHandle>*

Portable Request Format:

STATUS AllocateEvent(
OUT PHANDLE *EventHandle*
);

Parameters:

EventHandle - Specifies a 16-bit general purpose register or a pointer to a word in memory where the kernel will return a handle to the allocated event.

DeallocateEvent

An event object is destroyed with the **DeallocateEvent** kernel function. Once the event is deallocated, its corresponding handle becomes invalid and may no longer be used.

This function returns allocated resources to the system so that they may be recycled. If any threads are blocked on the event object, then they are automatically destroyed by the kernel.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   ax, <EventHandle>      ; (AX) = handle to event.
mov   dl, SYS_DEALLOCATE_EVENT
int   2dh
jc    Failure
```

Macro Instruction Format:

label **DeallocateEvent** *<EventHandle>*

Portable Request Format:

STATUS DeallocateEvent(
IN HANDLE *EventHandle*
);

Parameters:

EventHandle - Specifies a 16-bit general purpose register or word in memory containing a handle to an event to be deallocated.

SetEvent

An event object may be transfered to the *set* state by calling the **SetEvent** kernel function. Setting an event will cause all threads blocking on the event object to be rescheduled by the kernel.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   ax, <EventHandle>          ; (AX) = handle to event.
mov   dl, SYS_SET_EVENT
int   2dh
jc    Failure
```

Macro Instruction Format:

label **SetEvent** <*EventHandle*>

Portable Request Format:

STATUS SetEvent(
IN HANDLE *EventHandle*
);

Parameters:

EventHandle - Specifies a 16-bit general purpose register or word in memory containing a handle to an event to be set.

ClearEvent

An event object may be transfered to the *cleared* state by calling the **ClearEvent** kernel function. Clearing an event will not affect the status of thread blocking on the event object.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   ax, <EventHandle>          ; (AX) = handle to event.
mov   dl, SYS_CLEAR_EVENT
int   2dh
jc    Failure
```

Macro Instruction Format:

label **ClearEvent** <*EventHandle*>

Portable Request Format:

STATUS ClearEvent(
IN HANDLE *EventHandle*
);

Parameters:

EventHandle - Specifies a 16-bit general purpose register or word in memory containing a handle to an event to be cleared.

PulseEvent

An event object may be momentarily transfered to the *set* state by calling the **PulseEvent** kernel function. Pulsing an event will cause all threads blocking on the event object to be rescheduled by the kernel, but the event's status is not changed.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   ax, <EventHandle>          ; (AX) = handle to event.
mov   dl, SYS_PULSE_EVENT
int   2dh
jc    Failure
```

Macro Instruction Format:

label **PulseEvent** <*EventHandle*>

Portable Request Format:

STATUS PulseEvent(
IN HANDLE *EventHandle*
);

Parameters:

EventHandle - Specifies a 16-bit general purpose register or word in memory containing a handle to an event to be pulsed.

QueryEvent

The state of an event object may be inspected by calling the **QueryEvent** kernel function. The function returns the value TRUE if the event was set at the time of the call, and FALSE otherwise.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   ax, <EventHandle>          ; (AX) = handle to event.
mov   dl, SYS_QUERY_EVENT
int   2dh                        ; (AX) = 0 if clear, non-zero if set.
jc    Failure
```

Macro Instruction Format:

label **QueryEvent** *<EventHandle>, <EventStatus>*

Portable Request Format:

STATUS QueryEvent(
IN HANDLE *EventHandle***,**
OUT PUSHORT *EventStatus*
);

Parameters:

EventHandle - Specifies a 16-bit general purpose register or word in memory containing a handle to an event to be queried.

EventStatus - Specifies a 16-bit general purpose register or address of a word in memory where the kernel will return the status of the event.

WaitEvent

A thread may wait for an event to reach the *set* state by calling the **WaitEvent** kernel function. If the event is already in the *set* state at the time the request is issued, the thread will continue execution without waiting. If the event is in the *cleared* state at the time the request is issued, the thread will block until the event is pulsed or set, at which time the thread will resume execution.

If a thread is blocked on an event, the kernel immediately selects another thread in the system for execution. If no threads are available for execution, then the scheduler executes an idle loop waiting for a thread to be allocated or unblocked.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   ax, <EventHandle>          ; (AX) = handle to event.
mov   dl, SYS_WAIT_EVENT
int   2dh                        ; wait for pulse or set status.
jc    Failure
```

Macro Instruction Format:

label **WaitEvent** <*EventHandle*>

Portable Request Format:

STATUS WaitEvent(
IN HANDLE *EventHandle*
);

Parameters:

EventHandle - Specifies a 16-bit general purpose register or word in memory containing a handle to an event to be blocked on.

Mutex Object

The *mutex* object implements a **mut**ual **ex**clusion semaphore that is also MP-safe in multiprocessor systems. Mutex objects are allocated and deallocated in the same way that events are. They are acquired and released as needed to guard a critical section. A mutex is assigned a state, and must always be in either the *acquired* or *released* states.

Mutexes may be used to control exclusive access to any object that they are associated with. For example, the disk BIOS may be assigned a mutex that disallows more than one request at a time from being submitted to the BIOS. Similarly, a mutex can also be used to guard a software object, such as a data structure.

Mutexes should not be acquired at interrupt time, because an interrupt occurs during the context of some arbitrary thread. To acquire a mutex in response to an interrupt, a secondary thread may be created or unblocked on the fly at interrupt time so as to define a thread that can acquire the mutex and perform the guarded work.

AllocateMutex

A mutex object is created with the **AllocateMutex** kernel function. Once the mutex is allocated, it is automatically maintained by the kernel in response to service requests.

This function returns a handle to a mutex object that must be used by the caller in subsequent kernel services that operate on mutexes. The kernel initializes the mutex object to the *released* state.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   dl, SYS_ALLOCATE_MUTEX
int   2dh                    ; (AX) = handle to mutex.
jc    Failure
```

Macro Instruction Format:

label **AllocateMutex** *<MutexHandle>*

Portable Request Format:

STATUS AllocateMutex(
OUT PHANDLE *MutexHandle*
);

Parameters:

MutexHandle - Specifies a 16-bit general purpose register or a pointer to a word in memory where the kernel will return a handle to the allocated mutex.

DeallocateMutex

A mutex object is destroyed with the **DeallocateMutex** kernel function. Once the mutex is deallocated, it is no longer available to the caller, and the handle becomes invalid.

If a mutex is deallocated that has one or more threads still blocked trying to acquire it, those threads are automatically destroyed by the kernel.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   ax, <MutexHandle>      ; (AX) = mutex handle.
mov   dl, SYS_DEALLOCATE_MUTEX
int   2dh
jc    Failure
```

Macro Instruction Format:

label **DeallocateMutex** *<MutexHandle>*

Portable Request Format:

STATUS DeallocateMutex(
IN HANDLE *MutexHandle*
);

Parameters:

MutexHandle - Specifies a 16-bit general purpose register or a word in memory containing a handle to the mutex to be deallocated.

AcquireMutex

A mutex object is set to the *acquired* state with the **AcquireMutex** kernel function. Once the mutex is acquired, other threads will block on any further attempts to acquire the mutex until the first thread releases the mutex.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   ax, <MutexHandle>          ; (AX) = mutex handle.
mov   dl, SYS_ACQUIRE_MUTEX
int   2dh
jc    Failure
```

Macro Instruction Format:

label **AcquireMutex** *<MutexHandle>*

Portable Request Format:

STATUS AcquireMutex(
IN HANDLE *MutexHandle*
);

Parameters:

MutexHandle - Specifies a 16-bit general purpose register or a word in memory containing a handle to the mutex to be acquired.

ReleaseMutex

A mutex object is set to the *released* state with the **ReleaseMutex** kernel function. Once the mutex is released, the kernel returns to the caller. Additionally, if there is at least one thread blocked on the mutex waiting to acquire it, then the kernel leaves the mutex in the *acquired* state and unblocks the next thread waiting to acquire it. If there are no other threads waiting to acquire the mutex, then the kernel resets it to the *released* state.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   ax, <MutexHandle>          ; (AX) = handle to mutex.
mov   dl, SYS_RELEASE_MUTEX
int   2dh
jc    Failure
```

Macro Instruction Format:

label **ReleaseMutex** *<MutexHandle>*

Portable Request Format:

STATUS ReleaseMutex(
IN HANDLE *MutexHandle*
);

Parameters:

MutexHandle - Specifies a 16-bit general purpose register or a word in memory containing a handle to the mutex to be released.

Spinlock Object

The *spinlock* object implements a fast, atomic, method for guarding short sequences of instructions (less than 100 machine instructions) from being executed on any other processor. When a spinlock is acquired, any further attempts to acquire it cause the second attempt to spin in a loop, scanning the spinlock's value, until the lock is released by the acquiring thread of execution.

Spinlocks can be acquired at any time, including interrupt time. If a spinlock is acquired at task time or at interrupt time, and a second interrupt occurs that causes the same spinlock to be acquired, a deadlock will result. Therefore, if there is any chance that a spinlock will be acquired at interrupt time, then interrupts must be disabled *before* acquiring the spinlock, and reenabled sometime *after* the spinlock is released. Failure to observe this rule will cause system deadlock.

Spinlock objects are allocated and deallocated in the same way that events and mutexes are. They are acquired and released as needed to guard a critical section. A spinlock is assigned a state, and must always be in either the *acquired* or *released* states.

Caution: In the current implementation, spinlocks are identified by the storage location of the handle, not the handle itself. The contents of the storage location are the actual spinlock's state itself. Thus, a spinlock handle cannot be moved to another location, and then used as the same spinlock. This was done to improve the speed with which a spinlock could be acquired and released, in accordance with the goal of having a very "lightweight" kernel.

AllocateSpinlock

A spinlock object is created with the **AllocateSpinlock** kernel function. Once the spinlock is allocated, it is automatically maintained by the kernel in response to service requests.

This function returns a handle to a spinlock object that must be used by the caller in subsequent kernel services that operate on mutexes. The kernel initializes the mutex object to the *released* state.

Assembly Language Format:

none.

Macro Instruction Format:

label **AllocateSpinLock** <*SpinLockHandle*>

Portable Request Format:

STATUS AllocateSpinLock(
OUT PHANDLE *SpinLockHandle*
);

Parameters:

SpinLockHandle - Specifies a pointer to a word in memory where the kernel will return a handle to the allocated spinlock.

DeallocateSpinLock

A spinlock object is destroyed with the **DeallocateSpinLock** kernel function. Once the spinlock is deallocated, it is no longer available to the caller, and the handle becomes invalid.

If a spinlock is deallocated while other processors are attempting to acquire the lock, the lock remains in effect, and the other processors will spin indefinitely.

Assembly Language Format:

none.

Macro Instruction Format:

label **DeallocateSpinLock** <*SpinLockHandle*>

Portable Request Format:

STATUS DeallocateSpinLock(
IN PHANDLE *SpinLockHandle*
);

Parameters:

SpinLockHandle - Specifies a pointer to a word in memory containing a handle to the spinlock to be deallocated.

AcquireSpinLock

A spinlock object is set to the *acquired* state with the **AcquireSpinLock** kernel function. Once the spinlock is acquired, other processors will block attempting to acquire the lock until the spinlock is released.

Assembly Language Format:

none.

Macro Instruction Format:

label **AcquireSpinLock** <*SpinLockHandle*>

Portable Request Format:

STATUS AcquireSpinLock(
IN PHANDLE *SpinLockHandle*
);

Parameters:

SpinLockHandle - Specifies a pointer to a word in memory containing a handle to the spinlock to be acquired.

ReleaseSpinLock

A spinlock object is set to the *released* state with the **ReleaseSpinLock** kernel function. Once the spinlock is released, other processors may proceed to fight for the spinlock in an attempt to acquire it.

Assembly Language Format:

none.

Macro Instruction Format:

label **ReleaseSpinLock** <*SpinLockHandle*>

Portable Request Format:

STATUS ReleaseSpinLock(
IN PHANDLE *SpinLockHandle*
);

Parameters:

SpinLockHandle - Specifies a pointer to a word in memory containing a handle to the spinlock to be released.

Timer Object

The *timer* object implements an extremely fast, centralized, method for scheduling a source of interrupt-time execution at a specific interval of time in the future.

The timer object was designed to be extremely lightweight, especially acts of starting and stopping them, so that it could be used to time very high-speed events, such as packet retransmission timeouts in LAN transports. The design assumes that the creation of a timer can be a heavy event that is not performance critical. Because expiration timers almost never fire compared to the number of times that they are started and stopped, the starting and stopping of timers is made extremely fast.

When a timer expiration routine executes, it must not block, because the expiration routine is executed at interrupt time, not task time. If it is desired to block in response to a timer expiration, a task-time source of control can be gained by calling **AllocateThread** to create a parallel task-time environment, and then simply returning.

AllocateTimer

A timer object is created with the **AllocateTimer** kernel function. Once the timer is allocated, it is automatically maintained by the kernel in response to service requests.

At allocation time, the caller specifies a pointer to a function to be executed when and if the timer does expire. Additionally, a *context* value is specified by the caller that the kernel will pass to the expiration routine in the (BX) general register. In the portable request form, the interface automatically passes the context as a USHORT argument on the stack.

This function returns a handle to a timer object that must be used by the caller in subsequent kernel services that operate on timers. The kernel initializes the timer object to contain the context value and the pointer to the expiration function, so that these values need not be specified on every start/stop request.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   ax, <Context>          ; (AX) = context supplied to expiration rtn.
mov   bx, CGROUP
mov   cx, OFFSET CGROUP:ExpirationRtn ; (CX:BX) = routine address.
mov   dl, SYS_ALLOCATE_TIMER
```

```
int     2dh                     ; (AX) = handle to timer.
jc      Failure
```

Macro Instruction Format:

label **AllocateTimer** *<TimerHandle>, <Context>, <ExpirationRtn>*

Portable Request Format:

STATUS AllocateTimer(
OUT PHANDLE *TimerHandle***,**
IN USHORT *Context***,**
IN (* *ExpirationRtn*)**(USHORT)**
);

Parameters:

TimerHandle - Specifies a 16-bit general purpose register or a pointer to a word in memory where the kernel will return a handle to the allocated timer.

Context - Specifies a 16-bit general purpose register or a word in memory containing a value to be passed to the expiration routine when and if the timer expires.

ExpirationRtn - Specifies a far address of a function to be executed when the timer expires. For the macro format only, the 16-bit offset is assumed to be relative to the current CS segment register. The portable form of the request passes a far code pointer in the format generated by the high level language.

DeallocateTimer

A timer object is destroyed with the **DeallocateTimer** kernel function. Once the timer is deallocated, it is no longer available to the caller, and the handle becomes invalid.

If a timer object is deallocated when it is still running, then the timer is stopped automatically by the kernel before destroying it.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov     ax, <TimerHandle>       ; (AX) = handle to timer.
mov     dl, SYS_DEALLOCATE_TIMER
int     2dh
jc      Failure
```

Macro Instruction Format:

label **DeallocateTimer** *<TimerHandle>*

Portable Request Format:

STATUS DeallocateTimer(
IN HANDLE *TimerHandle*
);

Parameters:

TimerHandle - Specifies a 16-bit general purpose register or a word in memory containing a handle to the timer to be deallocated.

StartTimer

A timer object is set to the *running* state with the **StartTimer** kernel function. If a timer is already in the *running* state, then it remains in the *running* state, and only the expiration time is changed. If a timer is in the *stopped* state, then it is moved to the *running* state, and the expiration time is set to the current time plus the specified number of milliseconds in *DeltaTime*.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   ax, <TimerHandle>        ; (AX) = handle to timer.
mov   cx, <DeltaTime>          ; (CX) = milliseconds to expiration.
mov   dl, SYS_START_TIMER
int   2dh                      ; start the timer.
jc    Failure
```

Macro Instruction Format:

label **StartTimer** <*TimerHandle*>, <*DeltaTime*>

Portable Request Format:

STATUS StartTimer(
IN HANDLE *TimerHandle***,**
IN USHORT *DeltaTime*
);

Parameters:

TimerHandle - Specifies a 16-bit general purpose register or a word in memory containing a handle to the timer to be started.

DeltaTime - Specifies a 16-bit general purpose register or a word in memory containing the number of milliseconds to defer execution of the expiration routine.

StopTimer

A timer object is set to the *stopped* state with the **StopTimer** kernel function. If a timer is already in the *running* state, then its state is changed to *stopped*. If a timer is in the *stopped* state, then no operation is performed.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   ax, <TimerHandle>          ; (AX) = handle to timer.
mov   dl, SYS_STOP_TIMER
int   2dh
jc    Failure
```

Macro Instruction Format:

label **StopTimer** <*TimerHandle*>

Portable Request Format:

STATUS StopTimer(
IN HANDLE *TimerHandle*
);

Parameters:

TimerHandle - Specifies a 16-bit general purpose register or a word in memory containing a handle to the timer to be stopped.

Pool Object

The *pool* object is an extremely fast, centralized, near heap used for scratch memory allocation at any time, including interrupt time. While system pool is limited, it helps conserve code space by keeping only one memory allocator in memory, and it overcommits clients' memory needs, thereby saving data space.

Pool memory is allocated from only one segment, so that all pointers to objects carved out of pool have an implied and well-known segment/selector. The segment component of the address is passed around in the portable form of the requests.

System pool should be used in situations where either a very small amount of memory (under 128 bytes) is allocated to represent some other object (as the kernel does to allocate mutex objects, timer objects, and so on). It may also be used to allocate larger blocks that will quickly leave the system. For example, LAN packet drivers may safely acquire a 2kb block of system pool to represent an incoming packet, and upon completion of processing a millisecond later, the block may be released back to the system.

The kernel proper and all of the other Embedded DOS system components use system pool to build file objects, disk cache buffers, and so on.

The size of the system memory pool is configurable at system build time, and may be up to 65535 bytes in size, and is adjustable in OEM.INC and CONFIG.SYS through the SYSTEMPOOL= parameter. The minimum allocation size is also configurable at build time.

AllocatePool

A block of memory is allocated from system pool with the **AllocatePool** kernel function. Once the block is allocated, it may be used for read/write/execute access until deallocated with **DeallocatePool**.

All blocks allocated with this request are addressable in the same segment; therefore, the pointer to a block of pool memory is a near pointer relative to a well-known segment/selector. The system pool segment/selector is automatically returned in the portable form of this request to form a far pointer. The macro instruction form of this request returns the offset portion of the block in the (DI) general purpose register.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   ax, <BlockSize>          ; (AX) = size of requested block in bytes.
mov   dl, SYS_ALLOCATE_POOL
int   2dh                      ; (DI) = ofs FWA, allocated block.
jc    Failure
```

Macro Instruction Format:

label **AllocatePool** <*BlockSize*>

Portable Request Format:

STATUS AllocatePool(
IN USHORT *BlockSize***,**
OUT PVOID * *BlockPtr*
);

Parameters:

BlockSize - Specifies 16-bit general purpose register, a literal constant, or a pointer to a word in memory that contains the requested size of the block to be allocated in bytes. The kernel reserves the right to overallocate the block without notification.

BlockPtr - Specifies a pointer to a pointer that will be initialized by the kernel to point to the allocated block of storage. In the macro instruction form of the request, a near pointer is passed in (DI).

DeallocatePool

A block of allocated system pool is returned to the system with the **DeallocatePool** kernel function. Once the block is deallocated, it is no longer available to the caller, and the address of the block becomes invalid. The caller must assume that the system recycles the system pool at a very rapid rate, and upon return from this function, will have most likely already allocated the address to another system component.

The macro instruction form of this request assumes that the offset pointer to the block is already in the (DI) general purpose register when the macro is executed. The portable form of this request passes the entire pointer in the calling convention of the high-level language.

The kernel keeps a reference count on a block of pool memory. When the block is initially allocated, this reference count is set to 1. When the **DeallocatePool** request is executed, this reference count is decremented, and if the reference count drops to zero, the block is deallocated. Otherwise, the block remains locked in memory until the reference count is decremented with sufficient **DeallocatePool** requests to bring the reference count to zero. The reference count is incremented artificially with the **KeepPool** kernel function. This allows communicating kernel components to pass a data structure around without engaging in ownership protocol.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   di, <AllocatedBlockOffset>   ; (DI) = ofs FWA, block.
mov   dl, SYS_DEALLOCATE_POOL
int   2dh
jc    Failure
```

Macro Instruction Format:

label **DeallocatePool**

Portable Request Format:

STATUS DeallocatePool(
IN PVOID *BlockPtr*
);

Parameters:

BlockPtr - Specifies a pointer to a block of memory allocated with the **AllocatePool** kernel function. The address must not be normalized with segment arithmetic, because only the offset portion is examined by the kernel.

KeepPool

The **KeepPool** kernel function increments the reference count on a block of previously allocated pool memory, so that ownership is implicitly granted to the caller of the **KeepPool** function instead of the **AllocatePool** function.

The kernel keeps a reference count on a block of pool memory. When the block is initially allocated, this reference count is set to 1. When the **DeallocatePool** request is executed, this reference count is decremented, and if the reference count drops to zero, the block is deallocated. Otherwise, the block remains locked in memory until the reference count is decremented with sufficient **DeallocatePool** requests to bring the reference count to zero. The reference count is incremented artificially with the **KeepPool** kernel function. This allows communicating kernel components to pass a data structure around without engaging in ownership protocol.

In the macro instruction format of this request, the offset address of the block to be kept is passed in the (DI) general purpose register. In the portable form of this request, the far pointer is passed according to the calling conventions of the high level language.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   di, <AllocatedBlockOffset>   ; (DI) = ofs FWA, block.
mov   dl, SYS_KEEP_POOL
int   2dh
jc    Failure
```

Macro Instruction Format:

label **KeepPool** *<BlockPtr>*

Portable Request Format:

STATUS KeepPool(
IN PVOID *BlockPtr*
);

Parameters:

BlockPtr - Specifies a pointer to a block of memory allocated with the **AllocatePool** kernel function. The address must not be normalized as the kernel only uses the offset component of the address to locate the block of system pool.

Addressability Functions

The kernel provides functions through its interface that allow access to the various components of the DOS kernel and executive. This section describes those functions.

GetFsHelpAddress

Installable device drivers or even application programs may obtain the Embedded DOS file system helper API function address by calling this kernel function. This allows non-system code to interact with the file system helpers to register file system support and interact with the base block drivers, SDTEs, and SHTEs in the system that are associated with drives supported by those drivers.

The file system helper address is returned as a 16:16 address in the (DX:AX) register pair, and is typically stored in the calling FSD's header. This storing process is *not* provided by this function.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   dl, SYS_GET_FSHELP_ADDRESS
int   2dh                        ; (DX:AX) = FWA, FsHelp dispatcher.
```

Macro Instruction Format:

none.

Portable Request Format:

STATUS GetFsHelpAddress(
OUT PVOID * *FsHelpRtnPtr*
);

Parameters:

FsHelpRtnPtr - Specifies a pointer to a 32-bit storage location where the 32-bit pointer to the FSHELP dispatcher will be returned by this function. The caller can then use this value as a function pointer to indirectly call to the FSHELP dispatcher.

GetIoHelpAddress

Installable device drivers, file system drivers, and application programs may submit requests directly to the Embedded DOS I/O helper API, thereby bypassing the DOS process handling associated with the "current PSP". The address of the I/O helper API function dispatcher is returned by this function.

The I/O system helper address is returned as a 16:16 address in the (DX:AX) register pair, and may be stored anywhere by the calling program.

Upon return, the macro instruction call clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   dl, SYS_GET_IOHELP_ADDRESS
int   2dh                        ; (DX:AX) = FWA, IoHelp dispatcher.
```

Macro Instruction Format:

none.

Portable Request Format:

STATUS GetIoHelpAddress(
OUT PVOID * *IoHelpRtnPtr*
);

Parameters:

IoHelpRtnPtr - Specifies a pointer to a 32-bit storage location where the 32-bit pointer to the IOHELP dispatcher will be returned by this function. The caller can then use this value as a function pointer to indirectly call to the IOHELP dispatcher.

GetDosDataDs

Any system component may retrieve the DOSDATA segment value with the **GetDosDataDs** kernel function. This allows quick access to central data structures and pool objects.

In the assembly language and macro instruction forms, the DS register is initialized with the DOSDATA segment value.

Assembly Language Format:

```
mov    dl, SYS_USEDOSDATA_DS
int    2dh                    ; (DS) = DOSDATA segment.
```

Macro Instruction Format:

```
USEDOSDATA ds
```

Portable Request Format:

none.

Parameters:

none.

GetDosDataEs

Any system component may retrieve the DOSDATA segment value with the **GetDosDataEs** kernel function. This allows quick access to central data structures and pool objects.

In the assembly language and macro instruction forms, the ES register is initialized with the DOSDATA segment value.

Assembly Language Format:

```
mov    dl, SYS_USEDOSDATA_ES
```

```
int     2dh                     ; (ES) = DOSDATA segment.
```

Macro Instruction Format:

```
USEDOSDATA es
```

Portable Request Format:

none.

Parameters:

none.

GetDgroupDs

Any system component may retrieve the Embedded DOS DGROUP segment value with the **GetDgroupDs** kernel function. This allows quick access to central data structures.

In the assembly language and macro instruction forms, the DS register is initialized with the DGROUP segment value.

Assembly Language Format:

```
mov     dl, SYS_USEDGROUP_DS
int     2dh                     ; (DS) = system DGROUP segment.
```

Macro Instruction Format:

```
USEDGROUP ds
```

Portable Request Format:

none.

Parameters:

none.

GetDgroupEs

Any system component may retrieve the Embedded DOS DGROUP segment value with the **GetDgroupEs** kernel function. This allows quick access to central data structures.

In the assembly language and macro instruction forms, the ES register is initialized with the system DGROUP segment value.

Assembly Language Format:

```
mov     dl, SYS_USEDGROUP_ES
int     2dh                     ; (ES) = system DGROUP segment.
```

Macro Instruction Format:

```
USEDGROUP es
```

Portable Request Format:

none.

Parameters:

none.

GetPspDs

Any system component may retrieve the segment address of the current PSP with the **GetPspDs** kernel function. This allows quick access to the current DOS process in the I/O system.

In the assembly language and macro instruction forms, the DS register is initialized with the segment address of the current PSP.

Assembly Language Format:

```
mov     dl, SYS_USEPSP_DS
int     2dh                     ; (DS) = seg FWA, PSP.
```

Macro Instruction Format:

```
USEPSP ds
```

Portable Request Format:

none.

Parameters:

none.

GetPspEs

Any system component may retrieve the segment address of the current PSP with the **GetPspEs** kernel function. This allows quick access to the current DOS process in the I/O system.

In the assembly language and macro instruction forms, the ES register is initialized with the segment address of the current PSP.

Assembly Language Format:

```
mov   dl, SYS_USEPSP_ES
int   2dh                    ; (ES) = seg FWA, PSP.
```

Macro Instruction Format:

USEPSP es

Portable Request Format:

none.

Parameters:

none.

Named Objects

The kernel's named object service offers a method for cooperating applications to exchange information in a shared object name space. By associating ASCIIZ names with tokens or pointers, programs can tag kernel resources or user resources for sharing with other programs. This section describes the functions that implement the kernel's named object service.

AllocateObject

The **AllocateObject** kernel function associates a case-sensitive, user-supplied ASCIIZ string with a 32-bit token. Not interpreted by the system, the 32-bit token may be manufactured by the user to represent any data; it may be a far pointer, a file handle, a file pointer, an IP address, or even a handle to another system object. The object remains in the system until subsequently deallocated. This function returns a handle to the named object. The handle is used when manipulating the object with the other named object service API functions.

It is recommended, but not required, that the user partition the name space using the backslashes (\) normally used in filename space. The *recommended conventions* are as follows:

1. Start names with a backslash.
2. Follow the initial backslash with the name of a software component; i.e., "IP", "SMBSVR", "MSGPORT", or "QUEUE". Do not use software component names that begin with an underscore (_); these names are reserved for General Software's architectural expansion.
3. Follow the initial software component name with a backslash.
4. Follow the backslash with a unique name of your choice, which may include further partitioning with more backslashes.

Examples:

```
\MYPROG\Bill
\_MSGPORT\Toronto\MBX245\DATA
\_QUEUE\MyQueue
\DATA\ANY-OLD-DATA-TYPE
```

Once allocated, a named object persists in the system until it is deallocated, or the system maintaining the object is powered-down or rebooted.

Upon return, the assembly language software interrupt clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   dl, SYS_ALLOCATE_OBJECT
mov   cx, 1234h
mov   bx, 2345h                ; (CX:BX) = user value.
les   di, <AsciizName>         ; (ES:DI) = FWA, object's name.
int   2dh                      ; (AX) = handle to named object.
```

Macro Instruction Format:

none.

Portable Request Format:

STATUS AllocateObject(
IN UCHAR * *AsciizName***,**
IN PVOID *UserValue***,**
OUT PHANDLE *ObjectHandle*
);

Parameters:

AsciizName - Specifies a 32-bit segment:offset pointer to a case-sensitive, variable-length ASCIIZ string to be associated with the object.

UserValue - Specifies a 32-bit value to be associated with the object. This value may have meaning to the user application but is not inspected or interpreted by the system. This value may be retrieved with the **PointToObject** kernel function.

ObjectHandle - Specifies a 16-bit general purpose register or a pointer to a word in memory where the kernel will return a handle to the allocated named object.

AccessObject

The **AccessObject** kernel function accesses a named kernel object and returns a handle to the object. This handle is not the user-defined value associated with the object; instead, the handle is used as a token to pass to object-manipulation routines such as **PointToObject**.

Both **AllocateObject** and **AccessObject** increment a reference count on the object they return a handle for. If an object is allocated by one thread, and accessed by another, then the reference count on the object is two (2). If the first thread then deallocates the object, it loses its handle to the object, but the object is not deleted until the second thread deaccesses its handle to the object.

Upon return, the assembly language software interrupt clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   dl, SYS_ACCESS_OBJECT
les   di, <AsciizName>          ; (ES:DI) = FWA, object's name.
int   2dh                       ; (AX) = handle to named object.
```

Macro Instruction Format:

none.

Portable Request Format:

STATUS AllocateObject(
IN UCHAR * *AsciizName***,**
OUT PHANDLE *ObjectHandle*
);

Parameters:

AsciizName - Specifies a 32-bit segment:offset pointer to a case-sensitive, variable-length ASCIIZ string to search for in the system's object name space.

ObjectHandle - Specifies a 16-bit general purpose register or a pointer to a word in memory where the kernel will return a handle to the existing named object.

DeallocateObject

The **DeallocateObject** kernel function deaccesses a named kernel object, reducing the reference count on the object. Then, if the reference count is zero, the object is deleted from the system. If the reference count is nonzero, then the object remains in the system until the last reference to the object is removed, at which time the object is deleted.

Upon return, the assembly language software interrupt clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   dl, SYS_DEALLOCATE_OBJECT
mov   ax, <ObjectHandle>        ; (AX) = handle to named object.
int   2dh                       ; delete the object.
```

Macro Instruction Format:

none.

Portable Request Format:

STATUS DeallocateObject(
IN HANDLE *ObjectHandle*
);

Parameters:

ObjectHandle - Specifies a 16-bit handle to a named object as returned by the **AllocateObject** or **AccessObject** functions.

ReleaseObject

The **ReleaseObject** kernel function deaccesses a named kernel object, reducing the reference count on the object, but not deleting it. The **ReleaseObject** function may be called after allocating an object with **AllocateObject** to effectively leave the object in the system without any references to the object. It may also be used after calling the **AccessObject** function after access to the object is no longer necessary.

Both **AllocateObject** and **AccessObject** increment a reference count on the object they return a handle for. If an object is allocated by one thread, and accessed by another, then the reference count on the object is two (2). If the first thread then deallocates the object, it loses its handle to the object, but the object is not deleted until the second thread deaccesses its handle to the object.

Upon return, the assembly language software interrupt clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   dl, SYS_RELEASE_OBJECT
mov   ax, <ObjectHandle>          ; (AX) = handle to named object.
int   2dh                         ; release handle to the object.
```

Macro Instruction Format:

none.

Portable Request Format:

STATUS ReleaseObject(
IN HANDLE *ObjectHandle*
);

Parameters:

ObjectHandle - Specifies a 16-bit handle to a named object as returned by the **AllocateObject** or **AccessObject** functions.

LockObject

The **LockObject** kernel function acquires the specified object's mutex semaphore, causing all subsequent **LockObject** requests to block until a corresponding **UnlockObject** function is

executed. This mechanism allows multiple independent and cooperating programs to gain momentary exclusive access to the resources attached to the object without disabling interrupts or entering a system-wide critical section.

Upon return, the assembly language software interrupt clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   dl, SYS_LOCK_OBJECT
mov   ax, <ObjectHandle>          ; (AX) = handle to named object.
int   2dh
```

Macro Instruction Format:

none.

Portable Request Format:

STATUS LockObject(
IN HANDLE *ObjectHandle*
);

Parameters:

ObjectHandle - Specifies a 16-bit handle to a named object as returned by the **AllocateObject** or **AccessObject** functions.

UnlockObject

The **UnlockObject** kernel function releases the specified object's mutex semaphore, releasing exclusive access to the resources associated with the named object.

Upon return, the assembly language software interrupt clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov   dl, SYS_UNLOCK_OBJECT
mov   ax, <ObjectHandle>          ; (AX) = handle to named object.
int   2dh
```

Macro Instruction Format:

none.

Portable Request Format:

STATUS UnlockObject(
IN HANDLE *ObjectHandle*
);

Parameters:

ObjectHandle - Specifies a 16-bit handle to a named object as returned by the **AllocateObject** or **AccessObject** functions.

PointToObject

The **PointToObject** kernel function returns the 32-bit user value associated with a named object at the time it was created.

Upon return, the assembly language software interrupt clears the carry flag if the operation was successful, and sets it if the operation was not performed.

Assembly Language Format:

```
mov    dl, SYS_POINT_TO_OBJECT
mov    ax, <ObjectHandle>          ; (AX) = handle to named object.
int    2dh
```

Macro Instruction Format:

none.

Portable Request Format:

STATUS PointToObject(
IN HANDLE *ObjectHandle***,**
OUT PVOID * *UserValue*
);

Parameters:

ObjectHandle - Specifies a 16-bit handle to a named object as returned by the **AllocateObject** or **AccessObject** functions.

UserValue - Specifies a 16:16 pointer to a 32-bit user storage location where the user value associated with the named object is to be returned by the system.

Chapter 7

EXECUTIVE API

This chapter describes the executive of the EMBEDDED DOS 6-XL operating system. The purpose of the executive layer is to provide uniform, hardware-independent subsystems that automate common programming tasks.

Executive services are not implemented in the system image, DOS.SYS. Instead, these services are implemented in other external modules, and can be called from high-level language applications through bindings to files in the LIB directory. The following subsystems, prototype definition header files, and OBJ files are available for application use:

Message Ports:	INC\MESSAGE.H	LIB\MESSAGE.LIB
Queues:	INC\QUEUE.H	LIB\QUEUE.LIB

Executive services can be virtualized and remoted when appropriate drivers exist in the system. For example, with a suitable server and redirector driver loaded in the system, message ports could span a NetBIOS network containing many EMBEDDED DOS 6-XL nodes. The implementation or definition of the virtualization and remoting of executive services is beyond the scope of this text, although a major design consideration of all executive services is that they can be remoted in the future.

Access to Executive Services

Access to executive services is accomplished by linking with the appropriate LIB file (given above) to the user application.

The message and queue executive components are implemented entirely in their respective LIB files in the base system. Remoted versions of these components may involve device drivers, file system drivers, or other system objects.

Prototypes for executive services are defined in the system include directory (INC).

Message Port Services

The message port component of the executive offers a fast, fully-reentrant method for the exchange of variable-length data structures between cooperating threads. Message ports are implemented using the system-wide named object space, so that clients can refer to message ports by name.

Message ports are objects that serve as clearinghouses for send and receive requests. Multiple senders and receivers are allowed, and both send and receive operations can be asynchronous or synchronous, selectable on a request basis. Synchronous requests block until the operation has completed. Asynchronous requests return to the caller immediately, allowing the request processing to take place in parallel with the caller. At the caller's option, the completion of an asynchronous request can set a user-specified event object, so that user-defined threads can be notified when the request completes.

ExCreateMsgPort

A message port is created with the **ExCreateMsgPort** executive function. Once created, the port may be opened additional times for other clients, and may also service requests directly by the creating thread. The specified port name must be unique in the system's object name space; the function fails if a system object by the same name already exists.

Message ports are ephemeral objects that are automatically deleted when all handles to them are closed. The message component automatically maintains a reference count on message ports, incrementing the count on calls to **ExCreateMsgPort** and **ExOpenMsgPort**, and decrementing the count on calls to **ExCloseMsgPort**.

Portable Request Format:

STATUS ExCreateMsgPort(
IN UCHAR * *AsciizName***,**
OUT PHANDLE *MsgPortHandle*
);

Parameters:

AsciizName - Specifies the variable-length ASCIIZ string containing the name by which the message port is to be known. The message component automatically prepends the string "_MSGPORT\" to the name and inserts this extended form of the name in the system's object name space. User access to the message port through **ExOpenMsgPort** must not include the special system prefix manufactured by **ExCreateMsgPort**.

MsgPortHandle - Specifies a 16:16 (far) pointer to a 16-bit unsigned storage location where the executive will return a handle to the message port.

ExOpenMsgPort

An existing message port is accessed with the **ExOpenMsgPort** executive function. Once accessed, the port may be opened additional times for other clients as the need arises.

Message ports are ephemeral objects that are automatically deleted when all handles to them are closed. The message component automatically maintains a reference count on message ports, incrementing the count on calls to **ExCreateMsgPort** and **ExOpenMsgPort**, and decrementing the count on calls to **ExCloseMsgPort**.

Portable Request Format:

STATUS ExOpenMsgPort(
IN UCHAR * *AsciizName***,**
OUT PHANDLE *MsgPortHandle*
);

Parameters:

AsciizName - Specifies the variable-length ASCIIZ string containing the name of the message port to open. The message component automatically prepends the string "_MSGPORT\" to the name and uses the concatenated form to search the system's object name space for the message port object.

MsgPortHandle - Specifies a 16:16 (far) pointer to a 16-bit unsigned storage location where the executive will return a handle to the message port.

ExCloseMsgPort

An existing message port is closed with the **ExCloseMsgPort** executive function. Once all outstanding handles to a message port have been closed, the message port is deleted from the system.

Message ports are ephemeral objects that are automatically deleted when all handles to them are closed. The message component automatically maintains a reference count on message ports, incrementing the count on calls to **ExCreateMsgPort** and **ExOpenMsgPort**, and decrementing the count on calls to **ExCloseMsgPort**.

Portable Request Format:

STATUS ExCloseMsgPort(
IN HANDLE *MsgPortHandle*
);

Parameters:

MsgPortHandle - Specifies a 16-bit handle to a message port as returned by the **ExCreateMsgPort** and **ExOpenMsgPort** executive functions.

ExSendMsg

A message is sent to a message port with the **ExSendMsg** executive function. Messages are delivered reliably, as discrete packages. Each **ExSendMsg** request is satisfied by exactly one **ExReceiveMsg** request. If a message sent by ExSendMsg is too large to be received by **ExReceiveMsg**, the receiver is notified that the message was truncated.

If multiple senders are sending messages on the same message port, the messages are queued to the message port in the order in which they were received by the message component.

If multiple receivers are waiting to receive a message on the same message port, they are queued in the order in which they were received by the message component.

Portable Request Format:

STATUS ExSendMsg(
IN HANDLE *MsgPortHandle***,**
IN PUCHAR *Buffer***,**
IN USHORT *BufferLength***,**
IN USHORT *Flags***,**
IN HANDLE *Event*
);

Parameters:

MsgPortHandle - A 16-bit handle to a message port as returned by the **ExCreateMsgPort** and **ExOpenMsgPort** executive functions.

Buffer - A 16:16 pointer to a user buffer to be passed as a message.

BufferLength - A 16-bit unsigned quantity specifying the size of the buffer in bytes.

Flags - A 16-bit mask specifying options associated with the send request. Options may be combined by ORing them together. Caution: Do not use undefined flags in the API header file, as they are used internally; i.e., **MSG_FLAGS_USEREVENT** is used by the message component to remember whether the user specified an event to set; this is not for the caller to set.

MSG_FLAGS_WAIT - Instructs the **ExSendMsg** request to complete synchronously, blocking until the message is received with an **ExReceiveMsg** request. If this option is not selected, then the request is processed in parallel with the caller.

MSG_FLAGS_BUFFER - Instructs the **ExSendMsg** request to allocate a private temporary storage area and copy the contents of the specified buffer, so that the caller may reuse the storage for other purposes. If this option is selected, performance is somewhat impaired by the cost of the extra data copy, but it allows the caller to use a "fire and forget" approach to sending, eliminating the need for user buffer management.

Event - Specifies a 16-bit handle to an event to be set when the **ExSendMsg** request completes. If a zero handle is specified, then the executive will not set an event when the operation completes.

ExReceiveMsg

A message is received from a message port with the **ExReceiveMsg** executive function. Exactly one message is received by this request; multiple requests must be issued to read multiple messages. The executive performs no concatenation of messages; they are received exactly as sent.

If no messages are waiting to be received at the message port, then the message component will either block (waiting for an incoming message) if the **MSG_FLAGS_WAIT** bit is specified, or return and complete the receive asynchronously if not.

If multiple senders are sending messages on the same message port, the messages are queued to the message port in the order in which they were received by the message component.

If multiple receivers are waiting to receive a message on the same message port, they are queued in the order in which they were received by the message component.

Portable Request Format:

STATUS ExReceiveMsg(
IN HANDLE *MsgPortHandle***,**
IN PUCHAR *Buffer***,**
IN USHORT *BufferLength***,**
OUT PUSHORT *BytesTransferred***,**
IN USHORT *Flags***,**
IN HANDLE *Event*
);

Parameters:

MsgPortHandle - A 16-bit handle to a message port as returned by the **ExCreateMsgPort** and **ExOpenMsgPort** executive functions.

Buffer - A 16:16 pointer to a user buffer where the message component will return the received message.

BufferLength - A 16-bit unsigned quantity specifying the size of the buffer in bytes, and therefore the maximum size of a message that can be received by this request. If a received message is too large for the buffer, it is truncated.

BytesTransferred - A 16:16 pointer to a 16-bit storage location where the message component will return the number of bytes actually received into the buffer.

Flags - A 16-bit mask specifying options associated with the receive request. Options may be combined by ORing them together. Caution: Do not use undefined flags in the API header file, as they are used internally; i.e., the **MSG_FLAGS_USEREVENT** and **MSG_FLAGS_BUFFER** flags are not supported by this function.

MSG_FLAGS_WAIT - Instructs the **ExReceiveMsg** request to complete synchronously, blocking until a message is delivered to the message port with an **ExSendMsg** request. If this option is not selected, then the request is processed in parallel with the caller.

Event - Specifies a 16-bit handle to an event to be set when the **ExReceiveMsg** request completes. If a zero handle is specified, then the executive will not set an event when the operation completes.

Queueing Services

The queueing component of the executive offers a fast, fully-reentrant method for the management of ordered lists of data exchanged between cooperating threads. Queues are implemented using the system-wide named object space, so that clients can refer to queues by name.

Queues are objects that serve as clearinghouses for push, pop, and append requests. Multiple clients may push, pop, and append 32-bit data simultaneously. All requests complete synchronously.

ExCreateQueue

A queue is created with the **ExCreateQueue** executive function. Once created, the queue may be opened additional times for other clients, and may also service requests directly by the creating thread. The specified queue name must be unique in the system's object name space; the function fails if a system object by the same name already exists.

Queues are ephemeral objects that are automatically deleted when all handles to them are closed. The queue component automatically maintains a reference count on queues, incrementing the count on calls to **ExCreateQueue** and **ExOpenQueue**, and decrementing the count on calls to **ExCloseQueue**.

Portable Request Format:

STATUS ExCreateQueue(
IN UCHAR * *AsciizName***,**
OUT PHANDLE *QueueHandle*
);

Parameters:

AsciizName - Specifies the variable-length ASCIIZ string containing the name by which the queue is to be known. The queue component automatically prepends the string "_QUEUE\" to the name and inserts this extended form of the name in the system's object name space. User access to the queue through **ExOpenQueue** must not include the special system prefix manufactured by **ExCreateQueue**.

QueueHandle - Specifies a 16:16 (far) pointer to a 16-bit unsigned storage location where the executive will return a handle to the queue.

ExOpenQueue

An existing queue is accessed with the **ExOpenQueue** executive function. Once accessed, the queue may be opened additional times for other clients as the need arises.

Queues are ephemeral objects that are automatically deleted when all handles to them are closed. The queue component automatically maintains a reference count on queues, incrementing the count on calls to **ExCreateQueue** and **ExOpenQueue**, and decrementing the count on calls to **ExCloseQueue**.

Portable Request Format:

STATUS ExOpenQueue(
IN UCHAR * *AsciizName***,**
OUT PHANDLE *QueueHandle*
);

Parameters:

AsciizName - Specifies the variable-length ASCIIZ string containing the name of the queue to open. The queue component automatically prepends the string "_QUEUE\" to the name and uses the concatenated form to search the system's object name space for the queue object.

QueueHandle - Specifies a 16:16 (far) pointer to a 16-bit unsigned storage location where the executive will return a handle to the queue.

ExCloseQueue

An existing queue is closed with the **ExCloseQueue** executive function. Once all outstanding handles to a queue have been closed, the queue is deleted from the system.

Queues are ephemeral objects that are automatically deleted when all handles to them are closed. The queue component automatically maintains a reference count on queues, incrementing the count on calls to **ExCreateQueue** and **ExOpenQueue**, and decrementing the count on calls to **ExCloseQueue**.

Portable Request Format:

STATUS ExCloseQueue(
IN HANDLE *QueueHandle*
);

Parameters:

QueueHandle - Specifies a 16-bit handle to a queue as returned by the **ExCreateQueue** and **ExOpenQueue** executive functions.

ExPushQueue

A 32-bit user-architected value is pushed at the head of a queue with the **ExPushQueue** executive function, making it the next datum to be retrieved with the **ExPopQueue** function.

Portable Request Format:

STATUS ExPushQueue(
IN HANDLE *QueueHandle***,**
IN PVOID *UserValue*
);

Parameters:

QueueHandle - A 16-bit handle to a queue as returned by the **ExCreateQueue** and **ExOpenQueue** executive functions.

UserValue - An unarchitected, 32-bit user quantity that may be used by the caller as an integer, a pointer to storage, a function pointer, or some other object. The queue component does not inspect or interpret this value.

ExAppendQueue

A 32-bit user-architected value is appended to the tail of a queue with the **ExAppendQueue** executive function, making it the last datum in the queue to be retrieved with the **ExPopQueue** function.

Portable Request Format:

STATUS ExAppendQueue(
IN HANDLE *QueueHandle***,**
IN PVOID *UserValue*
);

Parameters:

QueueHandle - A 16-bit handle to a queue as returned by the **ExCreateQueue** and **ExOpenQueue** executive functions.

UserValue - An unarchitected, 32-bit user quantity that may be used by the caller as an integer, a pointer to storage, a function pointer, or some other object. The queue component does not inspect or interpret this value.

ExPopQueue

A 32-bit user-architected value is removed from the head of a queue with the **ExPopQueue** executive function. If the queue is empty when this request is made, then the **ExPopQueue** function returns with a failing status code.

Portable Request Format:

STATUS ExPopQueue(
IN HANDLE *QueueHandle***,**
OUT PVOID * *UserValue*
);

Parameters:

QueueHandle - A 16-bit handle to a queue as returned by the **ExCreateQueue** and **ExOpenQueue** executive functions.

UserValue - A 16:16 pointer to a 32-bit, unarchitected storage location where the datum is to be returned.

Chapter 8

DEBUGGING WITH DEBUG.SYS

This chapter describes the operation of the *kernel debugger*. The purpose of the debugger is to provide system-level, symbolic debugging facilities to the system programmer involved in developing, adapting, or maintaining components of the EMBEDDED DOS 6-XL system, installable device drivers, or file system drivers. The kernel debugger may also be used to debug application programs, but its facilities to debug user-level code are limited.

The kernel debugger loads as a device driver in CONFIG.SYS, as follows:

```
DEVICE=DEBUG.SYS
```

By default, the debugger is activated by pressing the ALT and the left SHIFT keys simultaneously. In this mode, the debugger uses the standard keyboard and video BIOS services to communicate with the PC's keyboard and screen.

The debugger can also be configured to run over a serial port so that it can be used even when no PC-compatible keyboard or screen is present, or when it is necessary to debug graphics applications. The following command line is used to redirect the debugger's output over a serial port (in this case, COM1):

```
DEVICE=DEBUG.SYS /PORT=1
```

Ideally, the kernel debugger is used for debugging activities such as gaining control when a program has been fully loaded, but before its first instruction gets executed (kernel symbol name *Launch*), or upon entry to a DOS function handler such as *DosWrite* (write bytes to a file). OEM-written add-ons to the kernel itself can be debugged with the kernel debugger (with full DOS symbols).

When active, the kernel debugger suspends all preemptive multitasking by setting *InDebugger* to a nonzero value. Routine ISR08, the timer tick handler, inspects this value and does not context switch if it is nonzero. This does not prevent ISRs from running, however. To stop ISRs, you need to program the interrupt controller from the debugger to mask-off the appropriate interrupts. The debugger does not do this by default because in most systems it is necessary to

keep the time-of-day clock going, and also to service keyboard interrupts so that the debugger can receive input from the keyboard or RS-232 port.

Kernel Symbols for DOS.SYS

When DOS.EXE is created by the MAKEFILE in GENERAL\EMBDOS, the corresponding DOS.MAP file is also created. This map is read by the MAKESYM utility to produce DOS.DBG, the symbol file read by the kernel debugger. This file must be in the root directory of the boot device in order to be recognized by DEBUG.SYS.

For a complete list of symbols in the kernel, consult the DOS.MAP file. If you are modifying a module of the system, such as OEM.ASM or SYSINIT.ASM, you may need to declare some symbols as PUBLIC in order to access them symbolically.

To make a data item PUBLIC, simply use the following general form in the DOS code:

```
          PUBLIC    foo             ; create debugger symbol.
foo       dw        1234            ; a PUBLIC data item.
```

Similarly, to make a procedure PUBLIC with the DefProc MACRO (as is used by all procedures in the DOS image), use the following form:

```
DefProc   MyRoutine, PUBLIC
          ...
EndProc   MyRoutine
```

Or, in the case of a FAR-callable routine:

```
DefProc   MyRoutine, FAR, PUBLIC
          ...
EndProc   MyRoutine
```

There are several symbols that are important in the DOS image that can aid your debugging of system-level activities, such as allocating threads and unblocking threads during interrupts to handle interrupt-time work at task time. The most important ones are given below:

BiosMutex - WORD containing the disk driver's handle to a mutex it uses for guarding access to INT 13h. You can inspect this mutex by using the DW command to get the handle in this word. Then you can use the MUTEX command to get the status of the mutex, as follows:

```
DEBUG: DW BiosMutex
0cb8:019c  0a66 0001 0a70 ...

DEBUG: MUTEX 0a66
Address   Flags  Status
0c8b:019c 0000   (Released)
```

BPBArray - WORD array containing offset pointers (relative to the segment address of the Disk device driver listed with the DEV command) to the current BPBs associated with each drive in the system. You can dump out this array with a DW

command and then, after using DEV to locate the segment address next to "Units=0x", display the actual BPB with the BPB command.

```
DEBUG: DW BPBArray
0cb8:0034  0044 006f 009a 00c5 ...

DEBUG: DEV
...
Address   Device   NextDevice  Strategy  ...
0c8b:0000 Units=03 ffff:ffff   0c8b:0281 ...

DEBUG: BPB 0cb8:0044
Address     BPS   SPT   NH     TS    HS ...
0c8b:0044   0200  0012  0002   0b40  0000 ...
```

CommandName - ASCIIZ BYTE array containing the initial COMSPEC variable used to load COMMAND.COM upon system initialization. You can dump this symbol out to determine if the name of COMMAND.COM has been properly set-up with SHELL= or hardcoded properly in SYSINIT.ASM:

```
DEBUG: DB CommandName
0094:00ae  41 3a 5c ...   A:\COMMAND.COM...
```

CommandTail - ASCIIZ BYTE array containing the initial command tail used to pass to the command interpreter when it is first loaded. Typically, this is "/P" for COMMAND.COM to recognize it is the primary interpreter. However, you may wish to name your program COMMAND.COM and pass it other arguments as required.

```
DEBUG: DB CommandTail
0094:0154  03 2f 50 0d 00 ... ./P..........
```

CritLevel - WORD containing the number of reasons why the kernel can't preemptively switch context to another thread in the system. *CritLevel* is automatically maintained by the system's thread object manager. Kernel function *EnterCriticalSection* increments this counter, while *LeaveCriticalSection* decrements it. If you believe that one of your threads is ready to but is not receiving control, break into the debugger and display this variable's contents (in this case, it is zero, meaning that preemptive scheduling is enabled):

```
DEBUG: DW CritLevel
0094:0098  0000 0001 1bd9 554e ...
```

DynDS - WORD containing the segment address of the EMBEDDED DOS 6-XL dynamic data segment, where system pool and other system data structures are allocated (this is not the address of the arena, where user program memory is allocated). Use the DW debugger command to display this address. It is important that this pointer not be corrupted by user programs. If it becomes corrupted, then system routines will not be able to locate the dynamic data segment.

```
DEBUG: DW DynDS
```

```
0094:008c  1176 1175 0000 0000 ...
```

ExecShell - Name of a routine in SYSINIT.ASM that runs the first program in the system, usually COMMAND.COM. You can set a breakpoint at this symbol to gain control before COMMAND.COM or your specified program receives control:

```
DEBUG: BP ExecShell
Breakpoint #0 set.

DEBUG: G
EMBEDDED DOS 6-XL Debugger [IN DOS] Breakpoint Trap
AX=0002  BX=00cc  CX=0001  DX=0040  SI=0028 ...
CS=0392  DS=0040  ES=1c1a  SS=1c1a  SP=03a6 ...
ExecShell (7ed6h):
0392:7ed6      push      ds

DEBUG:
```

IdleThread - Name of a routine in THREAD.ASM that runs when no other threads are runnable in the system. This routine always executes in the context of the idle thread at priority zero (0). You can set a breakpoint in the short loop in the IdleThread routine to gain control should all of your threads become blocked (this can also be accomplished at *OemIdle*, a routine called by IdleThread).

InDebugger - WORD containing the nesting level of the kernel debugger, so that it does not recursively enter itself during system tracing and other activities. Normally, when running system or user code, the value of this word is zero. When the debugger gains control, the value of this word is nonzero, causing the debugger to not breakpoint if a system event causes another trap or INT 3 (such as in an ISR). You can display the contents of this word with the DW debugger command:

```
DEBUG: DW InDebugger
0094:009a  0001 1bd9 554e 204c ...
```

Launch - Name of a label associated with the last JMP instruction in the DOS loader that transfers control to the user program when it is completely loaded and initialized. By tracing from that instruction, you will be able to step-through the initialization code in your program (even the C run-time libraries).

```
DEBUG: BP Launch
Breakpoint #0 set.

DEBUG: G
EMBEDDED DOS 6-XL Debugger [IN DOS] Breakpoint Trap
AX=0000  BX=0000  CX=0010  DX=3c10  SI=0800 ...
CS=0392  DS=3c10  ES=3c10  SS=4051  SP=07fe ...
launch (2ef6h):
0392:2ef7      jmpf      es:[003a]

DEBUG: T
EMBEDDED DOS 6-XL Debugger Single Step Trap
```

```
AX=0000  BX=0000  CX=0010  DX=3c10  SI=0800 ...
CS=3d86  DS=3c10  ES=3c10  SS=4051  SP=07fe ...
3d86:0014      mov       ah, 30h
```

Nunits - BYTE containing the number of disk units supported by the base disk device driver in the EMBEDDED DOS 6-XL system (this number does not include units associated with installable device drivers). By displaying this variable, you will be able to tell how many drives the base disk driver has allocated to the EMBEDDED DOS 6-XL system for access. The following example shows that the number of units is three (03).

```
DEBUG: DB Nunits
0c8b:000a  03 20 20 20 20 20 20 ...
```

OemKernelObjectInvalid - Name of a routine that is executed whenever your application passes an invalid object handle to an EMBEDDED DOS 6-XL kernel function. This breakpoint occurs in a routine called by the actual object manager. By tracing the RETN statement, you will cause it to return to the routine that detected the error. The invalid handle will be displayed in the AX register.

```
EMBEDDED DOS 6-XL Debugger [IN DOS] Breakpoint Trap
AX=1234  BX=00cc  CX=0001  DX=0018  SI=0028 ...
CS=0392  DS=1176  ES=1c1a  SS=1c1a  SP=038e ...
OemKernelObjectInvalid+0001h (86a0h):
0392:86a0      retn

DEBUG: T
EMBEDDED DOS 6-XL Debugger Single Step Trap
AX=1234  BX=00cc  CX=0001  DX=0018  SI=0028 ...
CS=0392  DS=1176  ES=1c1a  SS=1c1a  SP=0390 ...
SysSetEvent+0036h (6bcch):
0392:6bcc      sti
```

PassNumber - WORD containing the pass number (0, 1, 2, 3, or further) used by routines in CONFIG.ASM in parsing the CONFIG.SYS file. If your system breakpoints during processing of CONFIG.SYS, check this value to determine which pass was being run at the time of the breakpoint. In our example here, the pass number is 3, meaning the last pass.

```
DEBUG: DW PassNumber
0094:068b  0003 4f00 2056 4e00 2056 ...
```

SymTbl - WORD containing segment address of the segment where the symbol file, DOS.DBG, is loaded in memory. If this value is zero, then no symbol table is available. Of course, this symbol is invalid if the table is not loaded, but you can determine this symbol's address with symbols loaded, and then use it if the symbols are not loaded during initialization.

```
DEBUG: DW SymTbl
0094:181e  1c9c  4f44 2e53 4244 0047 ...
```

ToUser - Name of a label associated with the RETF 2 instruction that is the last system instruction in the INT 21h path. This instruction transfers control back to the user program after an INT 21h call has been executed by the system. By setting a breakpoint here, you can see the outcome of the next INT 21h call, including flags and registers, before EMBEDDED DOS 6-XL transfers back to the C Library or other calling code.

```
DEBUG: BP ToUser
Breakpoint #0 set.

DEBUG: G
EMBEDDED DOS 6-XL Debugger [IN DOS] Breakpoint Trap
AX=0a49  BX=0000  CX=1258  DX=0000  SI=1258 ...
CS=0392  DS=3b4c  ES=0000  SS=2b23  SP=331c ...
ToUser (728eh):
0392:728e      iret

DEBUG: T
EMBEDDED DOS 6-XL Debugger Single Step Trap
AX=0a49  BX=0000  CX=1258  DX=0000  SI=1258 ...
CS=20b0  DS=3b4c  ES=0000  SS=2b23  SP=3322 ...
20b0:0431      mov       ax, 049c
```

Setting Breakpoints

The EMBEDDED DOS 6-XL kernel debugger supports breakpoints by jamming an INT 3 instruction (one byte containing 0cch) into the first byte of the instruction you wish to breakpoint. Internally, the debugger maintains a list of breakpoint addresses and the replacement bytes at those addresses. This system is how most debuggers set breakpoints without hardware assist. It does not allow you to set breakpoints in ROM via the command line, since the ROM cannot be patched by the debugger with 0cch.

You can debug ROM code by placing an INT 3 instruction directly in the code path you wish to debug. When the INT 3 instruction is executed, control automatically transfers to the debugger so you can manipulate the system. Depending on how far the system has initialized, you may (or may not) have kernel symbols yet.

Stepping Through ROM Code

The kernel debugger also supports tracing through the TRAP bit in the processor flags, allowing you to single-step a code path, even though it is executing directly from ROM. This works through CPU hardware assistance and is available on all processors, including NEC V-series, Intel 80186 series, C&T 8680, and other Intel processors such as the 286, 386, 486, and Pentium CPUs.

Trapping on Illegal Writes to Data

The kernel debugger also supports a special modified trace that allows it to get control after each instruction is executed, so that it can examine any memory location to determine if it is modified. This is called a software watchpoint, and happens without hardware assistance other

than the CPU flags. While a watchpoint is enabled, code runs substantially slower (2 orders of magnitude longer) since it takes about 100 instructions to handle the trace interrupt in order to execute a single instruction.

The trace flag in the CPU also resets across an INT instruction. Thus, you won't be able to trace into BIOS calls such as INT 10h, for example.

Interacting With the Debugger

The kernel debugger is invoked through the console by pressing the keyboard's ALT and LEFT-SHIFT keys simultaneously. Once in the debugger, thread context switching is suspended, as is timer expiration, until resumed with the "G" (go) command. Additionally, the OEM can configure the debugger to communicate through a serial port rather than the console, if desired.

The debugger can receive programmed control from an INT 3 instruction in the system code, application code, or BIOS code. The debugger also provides PRINTF-style output formatting services through INT 2ch.

The debugger's command set is composed of two command classes: object displays and other commands. Nearly all commands are specified as a single abbreviated word such as "TIMER", "FSRP", or "EVENT", followed by an optional argument that is separated by a space, tab, or comma. Depending on the command's function, the address operand may default to the "next approriate address" or it may be required in the event that there is no "next appropriate address".

Object Display Commands

The most useful feature of the kernel debugger is its ability to display internal system structures in a human-readable format. The following structures are displayable by the kernel debugger:

- _ TIMER Kernel timer object
- _ EVENT Kernel event object
- _ MUTEX Kernel mutex object
- _ THREAD Kernel thread object
- _ ARENA System memory arena
- _ SPINLOCK Kernel spinlock object
- _ DEV Device driver chain
- _ FSD File system driver chain
- _ SHTE System handle table entry
- _ SDTE System drive table entry
- _ FSRP File system request packet
- _ IORP I/O request packet
- _ DDE FAT FSD disk directory entry
- _ BPB BIOS parameter block
- _ SPE System pool entry
- _ FFO FAT FSD file object
- _ FCB DOS file control block

TIMER Command

The TIMER command allows the developer to display the status of a particular timer object in the system, or all the timers in the system.

Command Syntax:

TIMER [*Address*]

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode address of an area of storage that is to be interpreted as a timer object. If the address does not point to a valid timer object, unpredictable output may result. If the address is not specified, then all the timers in the system are displayed.

Output Display:

```
Address   Abs Time  Delta     Flags Context  Expiration Routine
0ff6:0bb4 0003:a287 0000:000a 0003  0000     /Run ClockTick (00fah)
0ff6:0bc0 0003:af1b 0000:0c9e 0003  0000     /Run CacheTimeout (016fh)
0ff6:0bcc 0003:12be 0000:0041 0003  6387     /Run DfsTimerExpiration (67f0h)
```

EVENT Command

The EVENT command allows the developer to display the status of a particular event object in the system.

Command Syntax:

EVENT *Address*

Parameters:

Address - A required parameter that specifies the 16:16 real-mode address of an area of storage that is to be interpreted as an event object. If the address does not point to a valid event object, unpredictable output may result.

Output Display:

```
Address   Flags Status
0ff6:286a 0003  (Cleared)
```

MUTEX Command

The MUTEX command allows the developer to display the status of a particular mutex object in the system.

Command Syntax:

MUTEX *Address*

Parameters:

Address - A required parameter that specifies the 16:16 real-mode address of an area of storage that is to be interpreted as a mutex object. If the address does not point to a valid mutex object, unpredictable output may result.

Output Display:

```
Address   Flags  Status
0ff6:290e 0000   (Released)
```

THREAD Command

The THREAD command allows the developer to display the status of a particular thread object in the system, or all the threads in the system.

Command Syntax:

THREAD [*Address*]

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode address of an area of storage that is to be interpreted as a thread object. If the address does not point to a valid thread object, unpredictable output may result. If the address is not specified, then all the threads in the system are displayed.

Output Display:

```
Address   Flags Stack   SS:SP   BlockID Status
0ff6:4bac 0003  1644  157b:0364 0000    /Current
0ff6:4bc0 0001  16c4  16c4:03de 0000
```

SPINLOCK Command

The SPINLOCK command allows the developer to display the status of a particular spinlock in the system.

Command Syntax:

SPINLOCK *Address*

Parameters:

Address - A required parameter that specifies the 16:16 real-mode address of an area of storage that is to be interpreted as a spinlock object. If the address does not point to a valid spinlock object, unpredictable output may result.

Output Display:

```
Address   Flags  Status
0ff6:4286 0000   (Released)
```

ARENA Command

The ARENA command allows the developer to display the status of the memory arena managed by the INT 21h API functions **DosAllocSeg**, **DosFreeSeg**, and **DosSizeSeg**. It also permits formatting of a specific address as an arena header.

Command Syntax:

ARENA [*SegAddress*]

Parameters:

SegAddress - An optional parameter that specifies the segment address of an area of storage that is to be interpreted as an arena header. If the address does not point to a valid arena header, unpredictable output may result. If the address is not specified, then the system's memory arena is displayed.

Output Display:

```
Address   Fwdlink Baklink Size  Flags  PSP   Status
3503:0000 3d04    0000    0800  0001   163b  /Allocated
3d04:0000 3d15    3503    0010  0001   3d16  /Allocated
3d15:0000 3fb6    3d04    02a0  0001   3d16  /Allocated
3fb6:0000 0000    3d15    6049  0000   0000
```

DEV Command

The DEV command allows the developer to display the list of device drivers installed in the system, or a particular device driver header.

Command Syntax:

DEV [*Address*]

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode address of an area of storage that is to be interpreted as a device driver header. If the address does not point to a valid device driver header, unpredictable output may result. If the address is not specified, then the system's entire device driver chain is displayed.

Output Display:

```
Address   Device    NextDevice Strategy  Interrupt Attributes
1a44:0000 LOOPMAC$  0cfa:0000  1a44:004a 1a44:004b /Chr/Syn/V32/OpnClsRm
0cfa:0000 CON       0d10:0000  0cfa:0034 0cfa:0035 /Chr/Syn/OpnClsRm/29/In/Out
0d10:0000 AUX       0d25:0000  0d10:0034 0d10:0053 /Chr/OpnClsRm
0d25:0000 PRN       0d3a:0000  0d25:0034 0d25:0053 /Chr/OpnClsRm
0d3a:0000 NUL       0dee:0000  0d3a:0034 0d3a:0053 /Chr/OpnClsRm
```

```
0dee:0000 CLOCK$    0d4f:0000  0dee:0024 0dee:0025 /Chr/Clk
0d4f:0000 Units=03  ffff:ffff  0d4f:0281 0d4f:0282 /Block/RwCtl/Syn/OpnClsRm
```

FSD Command

The FSD command allows the developer to display the list of file system drivers installed in the system, or a particular file system driver header.

Command Syntax:

FSD [*Address*]

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode address of an area of storage that is to be interpreted as a file system driver header. If the address does not point to a valid file system driver header, unpredictable output may result. If the address is not specified, then the system's entire file system driver chain is displayed.

Output Display:

```
Address   File System  Next-FSD   Request    Version  FS-Help
0e24:0000 FAT          0fa1:0000  0e24:0040  0001     0464:56b0
0fa1:0000 DFS          0fa3:0000  0fa1:0014  0001     0464:56b0
0fa3:0000 UNC          ffff:ffff  0fa3:0034  0001     0464:56b0
```

SHTE Command

The SHTE command allows the developer to display all the entries in the system handle table (SHT) or a single system handle table entry (SHTE).

Command Syntax:

SHTE [*Address*]

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode address of an area of storage that is to be interpreted as a system handle table entry. If the address does not point to a valid SHTE, unpredictable output may result. If the address is not specified, then the system's entire system handle table is displayed.

Output Display:

```
Address   Object Flags Ref  Handle DevHdr    FSD  SDTE  Status
0ff6:2874 0002   0b12  0002 0000   0cfa:0000 0000 0000  /Dvc/DevH/Inh
0ff6:288e 0002   0b12  0002 0000   0cfa:0000 0000 0000  /Dvc/DevH/Inh
0ff6:28a8 0002   0b12  0002 0000   0cfa:0000 0000 0000  /Dvc/DevH/Inh
0ff6:28c2 0002   0312  0002 0000   0d10:0000 0000 0000  /Dvc/DevH/Inh
0ff6:28dc 0002   0312  0002 0000   0d25:0000 0000 0000  /Dvc/DevH/Inh
```

SDTE Command

The SDTE command allows the developer to display all the entries in the system drive table (SHT) or a single system drive table entry (SDTE).

Command Syntax:

SDTE [*Address*]

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode address of an area of storage that is to be interpreted as a system drive table entry. If the address does not point to a valid SDTE, unpredictable output may result. If the address is not specified, then the system's entire system drive table is displayed.

Output Display:

```
Address   Handle DevHdr     FSD   Unit  Status
0ff6:23f6 0000   0d4f:0000  0e24  00    Cwd="" /Fsd/Dev
0ff6:24be 0000   0d4f:0000  0e24  01    Cwd="" /Fsd/Dev
0ff6:2586 0000   0d4f:0000  0e24  02    Cwd="BIN" /Fsd/Dev
0ff6:2cd4 0000   0d3a:0000  0fa1  00    Cwd="" /Prefix="USA/WA"/Fsd/Net
```

FSRP Command

The FSRP command allows the developer to display the status of a file system request packet (FSRP) in the system.

Command Syntax:

FSRP *Address*

Parameters:

Address - A required parameter that specifies the 16:16 real-mode address of an area of storage that is to be interpreted as a file system request packet (FSRP). If the address does not point to a valid FSRP, unpredictable output may result.

Output Display:

```
Address   Status  Linkage    SHTE       Buffer     Path1      Path2
157b:0000 0815    0000:0100  0ff6:2918  6cf6:2d78  0ff6:2d86  0000:0000

          Flags   Count  Operation
          0000    10     FSD_OPENFILE
```

IORP Command

The IORP command allows the developer to display the status of an I/O request packet (IORP) in the system.

Command Syntax:

IORP *Address*

Parameters:

Address - A required parameter that specifies the 16:16 real-mode address of an area of storage that is to be interpreted as an I/O request packet (IORP). If the address does not point to a valid IORP, unpredictable output may result.

Output Display:

```
Address    Len Unit Cmd Status Linkage    MD  BufferAdr  Count Sector#
13f1:0000  b1  13   00  0100   13f1:0000  1c  dcc8:9c25  ab2c  569c:000e
```

DDE Command

The DDE command allows the developer to display the status of a DOS directory entry in the system.

Command Syntax:

DDE *Address*

Parameters:

Address - A required parameter that specifies the 16:16 real-mode address of an area of storage that is to be interpreted as a DOS directory entry (DDE). If the address does not point to a valid DDE, unpredictable output may result.

Output Display:

```
Address    File Name    Flags  Date  Time  Cluster  Size       Status
0ec6:2ef6  MYFILE  DAT  20     773a  14ec  269e     0000:0db7  /Archive
```

BPB Command

The BPB command allows the developer to display the status of a BIOS parameter block (BPB) in the system.

Command Syntax:

BPB *Address*

Parameters:

Address - A required parameter that specifies the 16:16 real-mode address of an area of storage that is to be interpreted as a BIOS Parameter Block (BPB). If the address does not point to a valid BPB, unpredictable output may result.

Output Display:

```
Address    BPS   SPT   NH    TS    HS         RS    NFAT  SPF   SPC  MDE   MD
0c06:0062  0200  0011  0004  a307  0000:0011  0001  02    0029  04   0200  f8
```

SPE Command

The SPE command allows the developer to display the status of a system pool entry in the system, or the entire system pool.

Command Syntax:

SPE *[Address*]

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode address of an area of storage that is to be interpreted as a system pool entry (SPE). If the address does not point to a valid SPE, unpredictable output may result. If no address is specified, then the entire system pool is displayed.

Output Display:

```
Address   Data      Refs  Length  Fwdlink  Type
0ff6:0bac 0ff6:0bb4 0001  04b0    1064     Untyped Data
0ff6:1064 0ff6:106c 0001  0080    10ec     I/O System Buffer
0ff6:10ec 0ff6:10f4 0001  0002    10f6     Mutex Object
0ff6:10f6 0ff6:10fe 0001  0002    1100     Mutex Object
0ff6:1100 0ff6:1108 0001  0200    1308     System Handle Table
0ff6:1308 0ff6:1310 0001  0034    1344     System Drive Table
0ff6:1344 0ff6:134c 0001  0002    134e     Mutex Object
0ff6:134e 0ff6:1356 0001  020c    1562     Disk Cache Entry
                                           ...
```

FFO Command

The FFO command allows the developer to display the status of a FAT File Object (FFO) in the system, or all of the FAT file objects in the system.

Command Syntax:

FFO *[Address*]

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode address of an area of storage that is to be interpreted as a FAT File Object (FFO). If the address does not point to a valid FFO, unpredictable output may result. If no address is specified, then all of the FFOs in the system are displayed.

Output Display:

```
Address   Flags  DOfs  Dir Sect   File Size  Dir Size   Start  Fwdlink
0ff6:2fe2 00f5   0140  0000:1506  0000:0031  0000:0000  2859   0000

          File Ptr  CClst  COfs  SHTE       Work  Status
          0000:0031 2859   0031  0ff6:2932  2d78  /Open/DirUpd/Fat16
```

SFD Command

The SFD command allows the developer to display the status of a system FCB descriptor (SFD) in the system, or all of the SFDs in the system.

Command Syntax:

SFD *[Address]*

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode address of an area of storage that is to be interpreted as a system FCB descriptor (SFD). If the address does not point to a valid SFD, unpredictable output may result. If no address is specified, then all of the SFDs in the system are displayed.

Output Display:

```
Address    Flags  FCB       Handle  PSP   Attr  Status
0eab:291c  0001   3d44:0000 0000    3d34  0020  /Locked
```

FCB Command

The FCB command allows the developer to display the status of an FCB or extended FCB in the system.

Command Syntax:

FCB *Address*

Parameters:

Address - A required parameter that specifies the 16:16 real-mode address of an area of storage that is to be interpreted as a file control block (FCB) or extended file control block (EFCB). If the address does not point to a valid FCB/EFCB, unpredictable output may result.

Output Display:

```
Address   Drive Filename     Cbn  Rs   FileSize  Date Time Sfd  Crn Rrn
3d44:0000 03    FCBFILE .DAT 0000 0000 0000:0000 0000 0000 0000 00  0000:0000
```

Other Commands

In addition to displaying system objects, the kernel debugger also allows the system developer to set breakpoints, step processor execution, dump memory in a variety of formats, and reboot the machine.

REBOOT Command

The REBOOT command allows the developer to reboot the system without removing power to the machine.

Command Syntax:

REBOOT

Parameters:

none.

Output Display:

none.

CONSOLE Command

The CONSOLE command allows the developer to redirect the debugger's input and output to another device. Available devices are:

CON - the system keyboard and video display monitor
COM1 - the first communications port at 3f8h
COM2 - the second communications port at 2f8h

Command Syntax:

CONSOLE *Device*

Parameters:

Device - A required parameter that specifies the new console to redirect debugger output to, and to redirect debugger input from.

Output Display:

none.

DOSDATA Command

The DOSDATA command allows the developer to inspect the major pointer values in the system.

Command Syntax:

DOSDATA

Parameters:

none.

Output Display:

```
DOS data DS             0ff6
Init-time DS            0094
Arena Base              3503
Memory Pool             0ff6:0bac
Current Process (PSP)   3d16:0000
System Process (PSP)    163b:0000
```

HELP Command

The "?" or HELP command allows the developer to display a summary of commands that are supported by the debugger.

Command Syntax:

HELP

? *(also accepted)*

Parameters:

none.

Output Display:

Short summary of available commands.

+ Command

The "+" command advances the instruction pointer (IP) by one byte. This command is useful when skipping over instructions.

Command Syntax:

+

Parameters:

none.

Output Display:

none.

- Command

The "-" command backs up the instruction pointer (IP) by one byte. This command is useful when an instruction should be reexecuted.

Command Syntax:

-

Parameters:

none.

Output Display:

none.

G Command

The G command allows the developer to resume execution from within the debugger.

Command Syntax:

G *[Address]*

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode address of a breakpoint to be set *before* execution begins at the current CS:IP address. If not specified, no breakpoint will be set.

Output Display:

none.

R Command

The R command allows the developer to display the contents of the register set associated with the currently executing thread. If at interrupt time, then the display shows the interrupt time register contents.

Command Syntax:

R

Parameters:

none.

Output Display:

```
EMBEDDED DOS 6-XL Debugger [IN BIOS] Copyright (C) 1996 General Software
AX=0093  BX=007a  CX=0001  DX=3d26  SI=001e  DI=0000  BP=03b6
CS=f000  DS=0040  ES=157b  SS=157b  SP=037e  IP=ebc3  NV UP EI NG NA PO ZR NC
f000:ebc3         cli
```

T Command

The T command allows the developer to trace through the current instruction and stop execution before the next one is executed. CALL and INT instructions are single-stepped by pushing into the called code; this command does not "step over" the instruction.

Command Syntax:

T

Parameters:

none.

Output Display:

```
EMBEDDED DOS 6-XL Debugger [IN BIOS] Copyright (C) 1996 General Software
AX=0093  BX=007a  CX=0001  DX=3d26  SI=001e  DI=0000  BP=03b6
CS=f000  DS=0040  ES=157b  SS=157b  SP=037e  IP=ebc3  NV UP EI NG NA PO ZR NC
f000:ebc3         cli
```

U Command

The U command allows the developer to display the contents of memory as a series of consecutive machine instructions. The instructions are formatted in a manner similar to most other debuggers' output.

By default, the U command unassembles at the current CS:IP address after a debugger break-in. Subsequent U commands display the next few instructions, and so on. Specifying a new address with the U command causes subsequent U commands to display the instructions following the last U command.

Command Syntax:

U *[Address]*

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode address of the first instruction to be decoded and displayed. If not specified, the display will start with the first instruction that follows the one last displayed in a U command.

Output Display:

```
RescheduleThread (620bh):
033f:620b         mov     di, [0068]
033f:620f         mov     [di+06], ss
033f:6212         mov     [di+04], sp
033f:6215         mov     [di+02], fffd
033f:6219         call    ScheduleThread (61b7h)
033f:621c         bkpt
033f:621d         retn
SysEnterCriticalSection (621eh):
033f:621e         push    ds
```

E Command

The E command allows the developer to change a series of consecutive 8-bit storage locations in memory.

Command Syntax:

E *Address Value1 [Value2 [Value3...]]*

Parameters:

Address - A required parameter that specifies the 16:16 address where the first byte in the sequence is to be stored. Subsequent bytes (if specified) are stored in consecutively higher bytes in memory.

Value1, Value2, etc. - A required set of one or more parameters that specify the hexadecimal 8-bit values to be stored at the specified address in memory.

Output Display:

none.

I Command

The I command allows the developer to issue a read to a byte-wide I/O port in the system. The value read from the port is displayed on the console.

Command Syntax:

I *IoAddress*

Parameters:

IoAddress - A required parameter that specifies the hexadecimal I/O port to read the 8-bit quantity from.

Output Display:

```
1a
```

O Command

The O command allows the developer to issue a write to a byte-wide I/O port in the system. The value written to the port is specified as the second parameter.

Command Syntax:

O *IoAddress Value*

Parameters:

IoAddress - A required parameter that specifies the hexadecimal I/O port to write the 8-bit quantity to.

Value - A required parameter that specifies the hexadecimal 8-bit value to write to the I/O port.

Output Display:

none.

BP Command

The BP command allows the developer to set an execution breakpoint at a specified address. Multiple breakpoints may be set at any given time.

Command Syntax:

BP *Address*

Parameters:

Address - A required parameter that specifies the 16:16 real-mode address of a breakpoint to be set.

Output Display:

```
Breakpoint #0 saved.
```

BC Command

The BC command allows the developer to clear an execution breakpoint.

Command Syntax:

BC *BreakpointNumber*

Parameters:

BreakpointNumber - A required parameter that specifies the number of the breakpoint to be cleared. The number of a breakpoint can be displayed with the BL command, and is displayed after the system processes a BP command.

Output Display:

```
Breakpoint #0 cleared.
```

BL Command

The BL command allows the developer to list the breakpoints that are currently active.

Command Syntax:

BL

Parameters:

none.

Output Display:

```
#0 - IoRead (4219h)
#1 - IoWrite (42aeh)
#2 - CacheTimeout (016fh)
```

WP Command

The WP command allows the developer to set a data watchpoint on a 16-bit storage area at the specified address. While the watchpoint is set, the processor enters *trace mode*, allowing the debugger to check the status of the storage area after the execution of each instruction to see if it has changed.

While it slows execution considerably (10x or more), a watchpoint can be very useful for finding instructions that are trashing memory.

Command Syntax:

WP *[Address]*

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode address of a 16-bit storage location in memory to be monitored. If not specified, the active watchpoint (if any) is cleared.

Output Display:

```
Watchpoint saved.
```

D Command

The D command allows the developer to display memory at the specified address, or at the address immediately following the last byte displayed with the last D, DB, DW, or DD command.

Command Syntax:

D *Address*

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode address of memory to be displayed in the default format. If not specified, then the address is assumed to be the address immediately following the last byte displayed by the last D, DB, DW, or DD command.

Output Display:

```
0040:0000 f8 03 00 00 00 00 00 00:bc 03 78 03 00 00 00 00  o.......L.x.....
0040:0010 7d 82 00 80 02 00 00 00:00 00 2c 00 2c 00 13 1f  }e.C......,.,...
0040:0020 13 1f 3f 35 0d 1c 03 2e:64 20 0d 1c 6f 18 75 16  ..>5....d...o.u.
0040:0030 74 14 0d 1c 62 30 6c 26:0d 1c 3f 35 0d 1c 01 00  t...b0l&..?5....
0040:0040 24 00 04 00 00 00 01 06:02 07 50 00 00 40 00 00  #.........P..@..
0040:0050 0b 18 00 00 00 00 00 00:00 00 00 00 00 00 00 00  <...............
0040:0060 07 00 00 b4 03 29 30 03:00 00 c8 00 b1 93 01 00  .....)0.....o..
0040:0070 00 00 00 00 00 01 81 00:14 14 14 14 01 01 01 01  ......u.........
```

DB Command

The DB command allows the developer to set the default memory display format to *bytes*, and then to display memory at the specified address, or at the address immediately following the last byte displayed with the last D, DB, DW, or DD command.

Command Syntax:

DB *Address*

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode address of memory to be displayed in bytes. If not specified, then the address is assumed to be the

address immediately following the last byte displayed by the last D, DB, DW, or DD command.

Output Display:

```
0040:0000 f8 03 00 00 00 00 00 00:bc 03 78 03 00 00 00 00  o.......L.x.....
0040:0010 7d 82 00 80 02 00 00 00:00 00 2c 00 2c 00 13 1f  }e.C.....,.,...
0040:0020 13 1f 3f 35 0d 1c 03 2e:64 20 0d 1c 6f 18 75 16  ..>5....d...o.u.
0040:0030 74 14 0d 1c 62 30 6c 26:0d 1c 3f 35 0d 1c 01 00  t...b0l&..?5....
0040:0040 24 00 04 00 00 00 01 06:02 07 50 00 00 40 00 00  #.........P..@..
0040:0050 0b 18 00 00 00 00 00 00:00 00 00 00 00 00 00 00  <...............
0040:0060 07 00 00 b4 03 29 30 03:00 00 c8 00 b1 93 01 00  .....)0.......o..
0040:0070 00 00 00 00 00 01 81 00:14 14 14 14 01 01 01 01  ......u.........
```

DW Command

The DW command allows the developer to set the default memory display format to *words*, and then to display memory at the specified address, or at the address immediately following the last byte displayed with the last D, DB, DW, or DD command. Data displayed in this format is in big-endian format (the numbers are real hexadecimal numbers that have been formatted by the debugger by swapping the low and high bytes of each word.)

Command Syntax:

DW *Address*

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode address of memory to be displayed in words. If not specified, then the address is assumed to be the address immediately following the last byte displayed by the last D, DB, DW, or DD command.

Output Display:

```
0090:0000   b3ea 6483 0004 4300 706f 7279 6769 7468
0090:0010   2820 2943 3120 3839 2039 6547 656e 6172
0090:0020   206c 6f53 7466 6177 6572 2000 2020 2020
0090:0030   2020 2020 2020 2020 2020 2020 2020 2020
0090:0040   4946 454c 0053 4346 5342 4200 4655 4546
0090:0050   5352 4300 554f 544e 5952 4400 5349 434b
0090:0060   4341 4548 4200 4552 4b41 5600 5245 4649
0090:0070   0059 5346 0044 4544 4956 4543 4300 4d4f
```

DD Command

The DD command allows the developer to set the default memory display format to *doublewords*, and then to display memory at the specified address, or at the address immediately following the last byte displayed with the last D, DB, DW, or DD command. Data displayed in this format is in big-endian format (the numbers are real hexadecimal numbers that have been formatted by the debugger by swapping the low and high bytes of each word, and the low and high words of each doubleword.)

Command Syntax:

DD *Address*

Parameters:

Address - An optional parameter that specifies the 16:16 real-mode address of memory to be displayed in doublewords. If not specified, then the address is assumed to be the address immediately following the last byte displayed by the last D, DB, DW, or DD command.

Output Display:

```
0090:0000   6483:b3ea 4300:0004 7279:706f 7468:6769
0090:0010   2943:2820 3839:3120 6547:2039 6172:656e
0090:0020   6f53:206c 6177:7466 2000:6572 2020:2020
0090:0030   2020:2020 2020:2020 2020:2020 2020:2020
0090:0040   454c:4946 4346:0053 4200:5342 4546:4655
0090:0050   4300:5352 544e:554f 4400:5952 434b:5349
0090:0060   4548:4341 4552:4200 5600:4b41 4649:5245
0090:0070   5346:0059 4544:0044 4543:4956 4d4f:4300
```

Programmed Output Formatting

The kernel debugger provides output-formatting services in the style of the C-language printf() library function for use in debugging an adaptation of EMBEDDED DOS 6-XL. These services are available through software interrupt INT 2ch, the KPRINTF macro, the DPRINTF macro, and the DISPLAYSTR macro (defined in MACROS.INC).

INT DBGINT (2Ch) Output Formatting API

The KPRINTF and DPRINTF macros format their output by calling INT 2ch (DBGINT) after setting up an appropriate stack frame. The general sequence of code to call the DBGINT service is as follows:

```
;       Define the formatting string in a segment that is contained in the
;       system's DGROUP group.  DBGMSG is the segment used by the KPRINTF &
;       DPRITNF macros for this purpose (this allows you to see the amount of
;       storage reserved for formatting strings in the link map.

DBGMSG  SEGMENT
addr    db      'Param1=$u, Param2=$x.', 0
DBGMSG  ENDS

;       First, push some 16-bit parameters.  If you need to display 8-bit
;       quantities, push a 16-bit object, and the output formatting routines
;       will only use the bottom 8 bits.  In this example, we will push two
;       objects, for a total of eight bytes of pointers on the stack.

        push    ax              ; first parameter, (AX) register.
        push    es:[di+2]       ; 2nd parameter, a memory word.

;       Now invoke the output formatting routines in the debugger.

        push    bp
```

```
mov     bp, sp
add     bp, 4           ; point BP to pushed arguments.
int     DBGINT          ; invoke formatting service.
dw      OFFSET DGROUP:addr ; ofs FWA, formatting string.
pop     bp
add     sp, 4           ; remove params from stack.
```

KPRINTF/DPRINTF Output Formatting Macros

Two macros are defined in DEFINES.INC for output formatting in the system itself. KPRINTF provides unconditional output, as is used by the system initialization code that displays the sign-on banner. DPRINTF provides conditional output only when the symbol DEBUG is nonzero; only in DEBUG builds of the system.

The PRINTF macros function very similarly to the C library's *printf* function. The remainder of this chapter discusses how to use the PRINTF macros and explains all of the formatting options. For this discussion, we will use DPRINTF as our macro name, but keep in mind that you can substitute KPRINTF wherever you see DPRINTF and get unconditional output formatting.

The basic DPRINTF macro syntax is as follows:

```
label   DPRINTF <fmtstr> [, <arg1 [,argn]>]
```

The label field is used by the assembler and can be used to transfer control to the DPRINTF statement. DPRINTF doesn't do anything with the label itself.

The formatting string, *fmtstr*, is any sequence of characters that your assembler will accept as a string. The angle brackets surrounding the formatting string are used by the assembler to group the string's characters together, even if the string contains commas and other separators. Be aware that the DPRINTF macro actually uses a DB statement in its expansion and surrounds the formatting string with single quotes; consequently, you must use two single quotes in succession whenever you wish to have one single quote printed in the string.

The formatting string is the basic template for the output to be performed. If no characters are present in the formatting string, then no output will be performed, regardless of the parameters specified in the argument list. The following is an example of a DPRINTF statement that prints "Hello World.\n":

```
DPRINTF <Hello World!\n>
```

Literal Specifications

Notice that, as with the C-library *printf* function, DPRINTF accepts literal characters, including the following:

\n	newline (CR/LF pair)
\r	carriage return (CR only)
\t	tab to next tab stop (1, 9, 17, etc.)
\b	bell character; beeps using BIOS
\\	display backslash character
\$	dollar sign (normally, $ is a formatting escape)

Format Specifications

Using DPRINTF to output strings with literals is a basic function. Of course, you probably also have data that needs to be formatted in many ways, because you will most lkely hav ethe data in BINARY form in a register or in memory. To display data such as BINARY words and strings using DPRINTF, we add two more components to the DPRINTF macro calls. First, we add an argument list after the print formatting string. Second, we introduce formatting specifiers inside the formatting string.

The DPRINTF macro accepts either one parameter or two parameters. If one parameter (enclosed in angle brackets) is specified, then that parameter is assumed to be a formatting string. If two parameters (both enclosed in their own angle brackets) are specified, separated by a comma, then the first parmaeter is assumed to be a formatting string, and the second parameter is assumed to contain a variable length list of arguments to be printed.

The argument list may contain zero, one, or more items to be formatted. The number of data items actually printed is a function of the print formatting string, and not the argument list. Hence the two parameters must be carefully and precisely coordinated.

Data items to be printed can be general processor registers, memory words, memory bytes, or strings in memory that are either fixed or variable length. Variable length strings may be terminated by a zero byte (00h) or a dollar sign ($).

The way that the data items are to be formatted is specified in your formatting string. All format specifications start with a dollar sign ($) and include a character after the dollar sign that indicates what type of formatting should be performed. The following table shows what formatting specifications are possible:

$c	prints bottom byte of word argument as raw character
$u	prints word argument as unsigned short number
$d	prints word argument as signed short number
$x	prints word argument as four hex digits
$lu	prints dword argument as unsigned long number
$ld	prints dword argument as signed long number
$lx	prints dword argument as eight hex digits
$b	prints bottom byte of word argument as two hex digits
$s	prints ASCIIZ string addressed by two word arguments
s	prints '$'-terminated string addressed by two word arguments
$s[n]	prints fixed length string of n characters

$c Format Specification

The following example displays the character in the AL register as an ASCII (raw) character:

```
DPRINTF <The character in AL is $c.\n>, <ax>
```

Similarly, to print the contents of a byte located in memory, you would declare the byte with the DB statement, but actually specify a WORD in the DPRINTF argument list:

```
OurChar DB      'A'             ; character to print.

        DPRINTF <OurChar contains $c.\n>, <word ptr OurChar>
```

$b Format Specification

To display the contents of an 8-bit location in hexadecimal, the $b format specification must be used. While $c printed the quantity as a single character, the $b specification interprets the byte as a BINARY number from 0-255, and then formats the number in base 16. The result is two digits that can take on values from 00 to ff.

Because an 8-bit quantity is being displayed, the same rules for passing 8-bit arguments as defined in the $c formatting section apply here also. Therefore, you must fool the assembler and actually pass a word to the DPRINTF macro to satisfy the macro expansion.

The following is an example call to display the contents of the CL register in hexadecimal format:

```
        DPRINTF <The hex value in CL is $b.\n>, <cx>
```

$x Format Specification

The $x format specification is similar to $b, except that a 16-bit quantity is displayed in hexadecimal format instead of an 8-bit quantity.

The following example shows how to print the contents of the SI register using the DPRINTF $x format specification:

```
        DPRINTF <The hex value in SI is $x.\n>, <si>
```

Similarly, you can print the contents of a memory word with some form of the following example:

```
MyWord  DW      12345           ; a word in memory.

        DPRINTF <MyWord in hex is $x.\n>, <MyWord>
```

Don't forget that the output is printed in hexadecimal. Therefore, this example doesn't print "12345", because that is the decimal value of the contents of MyWord. Instead, the $x format specification will display this value as "3039", because it is printed in base 16, not base 10.

$u Format Specification

The $u format specification is similar to $x, except that the 16-bit quantity is displayed in decimal (base 10) format instead of hexadecimal (base 16). The following example shows how to print the contents of the BX register using the DPRINTF $u format specification:

```
        DPRINTF <The value in BX is $u.\n>, <bx>
```

$d Format Specification

The $d format specification is similar to $u, except that the 16-bit quantity is displayed in integer decimal (base 10) format instead of unsigned base 10 format. Thus, if the top bit in the word is set, then the value is treated as a 2's complement negative number, and is displayed after a minus sign to indicate that it is a negative quantity. Keep in mind that 16-bit words can hold positive nubmers in the range 0 to 32767 and negative numbers -1 to -32768. Thus, if you store the unsigned value 32768 in a word and then format it with the $d format specification, it will be printed as -1.

The following example shows how to print the contents of the AX register using the DPRINTF $d format specification:

```
DPRINTF <The value in AX is $d.\n>, <ax>
```

$lx Format Specification

The $lx format specification is similar to $x, except that a 32-bit quantity is displayed in hexadecimal format instead of a 16-bit quantity.

The following example shows how to print the contents of the 32-bit quantity in the register pair (DX:AX) using the DPRINTF $lx format specification:

```
DPRINTF <The hex value in DX:AX is $lx.\n>, <dx, ax>
```

$lu Format Specification

The $lu format specification is similar to $lx, except that the 32-bit quantity is displayed in decimal (base 10) format. The following example shows how to print the contents of the 32-bit quantity represented by the CX:BX register pair using the DPRINTF $lu format specification:

```
DPRINTF <The 32-bit value in CX:BX is $lu.\n>, <cx, bx>
```

$ld Format Specification

The $ld format specification is similar to $lu, except that the 32-bit quantity is displayed in integer decimal (base 10) format instead of unsigned format. Keep in mind that 32-bit dwords can hold positive numbers in the range 0 to 2^31-1 and negative numbers -1 to -2^31. Thus, if you store the unsigned value 4292967295 (the decimal equivalent of 2^31) in a longword and then format it with the $ld format specification, it will be printed as -1.

The following example shows how to print the contents of the DX:AX register pair using the DPRINTF $ld format specification:

```
DPRINTF <The 32-bit value in DX:AX is $ld.\n>, <dx, ax>
```

$s Format Specification

The $s format specification allows you to display strings within your output. There are three forms of this format specification.

First, without any other modifiers, $s will simply output an ASCIIZ string (a variable length string containing a zero-byte at the end of it).

Second, by placing a dollar sign directly after the '$s', you can instruct DPRINTF to display a variable length string terminated by a dollar sign instead of a zero byte. This string format is commonly found in DOS programs that use DOS function 09h, **DosConStrOutput**.

Third, you can use a format specification derived from the general form, '$s[n]', where the brackets tell DPRINTF that the base 10 number inside the brackets is the length of the string.

Regardless of how the string is terminated or how long it is, its address must be fully specified in the argument list. Because strings are in general larger than one word, the address of the string instead of the string itself is passed in the argument list. And because the string may be located in any segment, both the segment and offset components of the string's address must be specified. As a result, you must specify two arguments (not one) in your argument list for every string formatter you use. The first argument is the segment address of the string, and the second argument is the offset address relative to that segment.

Finally, processors designed before the 80286 did not have a PUSH immediate instruction. As a consequence, it is not possible to push a segment value or an offset value of something without first putting it into a register, and then pushing the contents of the register. There is simply no instruction for pushing immediate data. We get around this processor limitation by simply storing string addresses in memory words or processor registers, and then passing the memory words or registers to DPRINTF's argument list.

The following example shows how to print the contents of a string that is pointed to by the ES:DI register pair (there is nothing magic about ES and DI, it could have been AX:BX or BP:DX). The string is zero-byte terminated.

```
        DPRINTF <The string contains "$s".\n>, <es, di>
```

This next example shows how to print the contents of a string that is statically declared as a memory array of bytes using the DB directive. The string is zero-byte terminated. We assume that MyString is in the data segment (addressable with DS).

```
MyString DB     'Hi there.', 0

        lea     ax, DGROUP:MyString
        DPRINTF <The string is $s.\n>, <ds, ax>
```

If MyString had been assembled in the code segment, then the argument list <cs, ax> would have been appropriate. Remember that the LEA instruction is only one of several ways to get the address of MyString into the AX register. Another would be to use the MOV instruction with the OFFSET operator:

```
MyString DB     'Hi there.', 0

        mov     ax, OFFSET DGROUP:MyString
        DPRINTF <The string contains "$s".\n>, <ds, ax>
```

Notice that we used the syntax "DGROUP:MyString". This indicates to the assembler that the offset component of the address is to be calculated relative to the group or segment called "DGROUP. DGROUP is not a magic name to the assembler, it is simply the most common name for the group of segments in the data group that most people use. If MyString had been in the code segment, and you were using CGROUP as a code group, then you would substitute "CGROUP" for "DGROUP" in the above example.

s Format Specification

So far, we have seen ways to print zero-byte terminated (ASCIIZ) strings with many different kinds of addressing. The s format specification allows the same addressing methods to be used, but simply defines the end of the string to be printed as the first occurrance of a dollar sign ($) in the string instead of a zero byte. here is an example where a $-terminated string is printed using the LEA-style addressing:

```
MyString DB     'Hi there.$'

         lea     ax, DGROUP:MyString
         DPRINTF <The string contains "$s$".\n>, <ds, ax>
```

$s[n] Format Specification

Still a third way to define the end of a string to be printed with $s is to include the syntax, [n], following the $s specification. This tells DPRINTF that the string is exactly n characters long, and that no characters in the string are to be treated as end-of-string terminators.

The following example shows how a fixed-length string can be printed with the DPRINTF macro:

```
MyString DB     'abcdefghijklmnop'

         lea     ax, DGROUP:MyString
         DPRINTF <The string contains "$s[16]".\n>, <ds, ax>
```

Chapter 9

TROUBLESHOOTING

If you need help deciphering a run-time error message printed by EMBEDDED DOS 6-XL or COMMAND.COM, see the "Error Messages" section in this chapter.

For other problems, we have maintained a list of technical support problems and their solutions, and have listed them under "Advanced Troubleshooting" in this chapter.

Advanced Troubleshooting

This section covers problems by application category. For example, all common RS-232-related problems have been grouped together so that you can make sure that you have addressed a set of potential hazards in your application.

COMMAND.COM Issues

Two COMMAND.COM command interpreters are provided; the standard one in GENERAL\UTIL and an auxilliary one in GENERAL\TINYCMD that uses only 12KB of memory. The tiny COMMAND.COM has reduced functionality but offers space savings. The standard COMMAND.COM has typical desktop functionality except piping and redirection, which are generally not useful in embedded applications.

The standard COMMAND.COM shell is a large model program that will swap itself to XMS if HIMEM.SYS is loaded in CONFIG.SYS. If you are running out of memory when running applications, check to see that HIMEM.SYS is loaded. The resident portion of COMMAND.COM is approximately 4 KB.

There are subtle syntax differences between these COMMAND.COM interpreters. The tiny COMMAND.COM interpreter has more restrictions on syntax than the larger one.

CONFIG.SYS Issues

During boot-up, the EMBEDDED DOS 6-XL system initialization sequence reads the CONFIG.SYS file for configuration. Parameters such as BUFFERS= or FILES= have no effect in an EMBEDDED DOS 6-XL system because allocation of the equivalent structures in EMBEDDED DOS 6-XL happens dynamically. Do not expect improved performance or expanded capabilities by manipulating these obsolete parameters. See the CONFIG.SYS reference chapter for a list of obsolete parameters.

CONFIG.SYS is parsed in several passes, so that UMB statements can be processed after *all* device drivers are loaded, and so on. The SYSTEMPOOL, STACKS, and STACKSIZE statements are processed *first*, so that they may be allocated before the device drivers (since device drivers may need pool or stacks to run properly).

There is no limit to the size of a CONFIG.SYS file. In generic DOS, a 64KB limitation is imposed.

Some third-party device drivers loaded with DEVICE= in CONFIG.SYS may not run in the EMBEDDED DOS 6-XL environment if they inspect or modify internal generic DOS structures. EMBEDDED DOS 6-XL necessarily has different internal structures used to provide the same functionality with full reentrancy. If a third-party driver has problems running under EMBEDDED DOS 6-XL, consult with the vendor to determine if undocumented DOS structures are being inspected or edited.

File System Issues

The EMBEDDED DOS 6-XL FAT file system driver is an advanced, bimodal system that offers a resiliant hardened mode as well as a cached mode for superior performance.

If you have the cache enabled (with VERIFY=OFF), then data written to cache by an application program may not be flushed properly if the system is halted or rebooted. To flush the data, call the DOSDISKRESET INT 21h function, or execute the SYNCH command in the standard COMMAND.COM command interpreter.

Do not attempt to use EMBEDDED DOS 6-XL disk utilities under another DOS system. These utilities assume that certain EMBEDDED DOS 6-XL services are available to perform their work.

Do not attempt to use generic DOS utilities (from MS-DOS, DR-DOS, or otherwise) on an EMBEDDED DOS 6-XL system; they may assume that internal structures are available that are in fact different from their undocumented generic DOS counterparts.

Do not use EMBEDDED DOS 6-XL with disk doublers or doublespace drivers; they inspect and modify undocumented DOS structures not available in EMBEDDED DOS 6-XL.

Do not use EMBEDDED DOS 6-XL with file systems previously managed with disk doublers or double-space drivers; the EMBEDDED DOS 6-XL file system does not support them, and neither do the disk utilities.

If you have trouble running CHKDSK, FORMAT, DISKCOMP, DISKCOPY, FDISK, or other disk utility programs, make sure that the EMBEDDED DOS 6-XL system has been configured for their use. The OPTION_DISK_IOCTLS option must be enabled for them to work properly.

Multitasking Issues

Most embedded systems will have satisfactory performance with the standard PC 18.2 Hz system timebase used for preemptive scheduling and timing. This rate works out to about 55ms per timeslice, and 55ms granularity on timeout values.

If you have several threads running in your application that need to access the same nonreentrant routines or the same data structures, you need to use an EMBEDDED DOS 6-XL mutex object (mutual exclusion semaphore) to ensure that only one thread uses the code or data at any given time. Functions such as printf, fopen, or spawnlp are nonreentrant, and must be guarded by mutexes if you are to make simultaneous calls to them.

When using interrupt service routines, be aware that the primary 8259 PIC in standard PC/AT systems assigns the highest priority to the timer tick (IRQ 0), making it possible for this interrupt to preempt your ISR when you re-arm the 8259 with an EOI. Thus, it is possible for your ISR to be treated as an extension of the currently-running thread, and for it to be preempted by the scheduler for a large period of time (i.e., 55ms based on an 18.2 Hz tick rate). You may use the IRQPRIORITY= statement in CONFIG.SYS to rotate the priority assignments on the first PIC to assign a lesser interrupt priority for rescheduling to avoid having your ISRs preempted, or you may wish to rearm the 8259 only after you have performed all the work necessary to service the interrupt.

Critical sections are different than sections guarded with mutexes. A mutex is associated with a specific object that needs exclusion, while critical sections disable preemption in the scheduler. Thus, if you wish to completely disable context switching based on timer ticks in a segment of code, EnterCriticalSection may be called to lock other threads out. Acquiring a mutex, however, does not affect all threads in the system; only those threads that might also acquire the mutex. Using mutexes over critical sections wherever possible will improve the system's task-time response to external events.

Performance Issues

Disk performance is a function of the performance of the underlying BIOS, the performance of the file system cache, and the operating mode of the file system. High performance can be effected by enabling the cache through setting VERIFY=OFF; reduced performance (at the gain of resiliancy) will result if the cache is disabled by setting VERIFY=ON.

Having SMARTDRV.EXE running is key to good disk performance. SMARTDRV.EXE works as an external cache to the file system's cache, keeping large buffers in XMS memory. While the internal cache (controlled with VERIFY) should be enabled, SMARTDRV will augment its performance considerably.

The CONFIG.SYS parameters, CACHESIZE=, CACHETTL=, and CACHEFLUSH=, all contribute to tuning the file system's performance. CACHESIZE is used to declare the maximum number of 512-byte cache blocks that can be active in the system at any given time. CACHETTL is used to control the time-to-live for each cache block (a block's age is computed

in milliseconds and restarts when it is read or written). Blocks reach the CACHETTL age and then are removed from the system. The CACHEFLUSH parameter controls the age (in milliseconds) at which a cache block must be flushed to disk if it contains data not previously written to the media. CACHEFLUSH should be less than CACHETTL, although the file system write-behind algorithm makes sure that it does not discard cache buffers without flushing them.

With the cache off (VERIFY=ON), the cache is still used, but only in a read mode. Writes are written through the cache blocks to disk, so that it is impossible to have live data in the cache that has not been flushed to disk.

To improve INT 21h function performance, consider setting the DOS BREAK flag to OFF. By default, BREAK is ON, causing the INT 21h function dispatcher to issue an I/O request to the CON device driver to see if a ^C has been entered by the console operator. If so, the ^C exception handler is executed. This is how typing ^C on the console when running a program from COMMAND.COM causes the current program to terminate. By turning this mechanism OFF, the overhead associated with calling the CON device driver on every I/O is eliminated. For applications where many small I/O requests are being processed by the INT 21h system, performance can be significantly improved.

RAM Cram Issues

If your application is large enough that you are running out of low memory in an EMBEDDED DOS 6-XL system, consider eliminating the standard COMMAND.COM; most embedded systems do not need COMMAND.COM. The standard command interpreter consumes space that would otherwise be available to your program.

Additional space (32KB) may be found by not allowing a debug DOS system to load the DOS.DBG file; either renaming the file or removing it from the boot device will allow this segment to be returned to the system memory arena, where it can be used by applications.

You can control how much memory is used for the file system cache, open file objects, and other system objects with the SYSTEMPOOL= statement in CONFIG.SYS. Break into the kernel debugger during your system's peak activity and type the POOL command for a list of pool objects. at the end of the listing is most likely an "unallocated pool" area of some specified size. This is the amount of system pool that is not used by your running system.

Reentrancy Issues

More complex C library functions are not reentrant; they actually use static data in their data segment to maintain temporary results. Printf, sprintf, malloc, free, fopen, and fclose are examples of library functions that are nearly always written in a non-reentrant fashion by your compiler vendor. Other functions, such as strcpy, strcmp, memmove, and atoi, for example, are likely to be reentrant.

If you experience unexpected crashes in your application when using the C library functions from multiple threads simultaneously, consider guarding them with an AcquireMutex call before calling the function, and calling ReleaseMutex after the call. You may wish to create a wrapper function, such as x_fopen, for your fopen function that performs the Acquire/Release Mutex calls around the actual C library function call.

You may make simultaneous calls to the INT 21h API from multiple threads. Generic DOS does not allow this. When calling INT 21h from an interrupt service routine, you should make sure that the interrupted thread is not also calling an INT 21h that would be blocked for the same reason as your ISR-time INT 21h call would block. If you suspect that this would be a problem (i.e., calling DOSREAD on the same file handle), then you should spin-off a thread in the ISR to do the work at task time, or alternatively pulse an event that a waiting thread is blocked on to do the ISR-related work.

RS-232 Communications Issues

EMBEDDED DOS 6-XL initializes the serial ports in the AUX device driver (see AUXDEV.ASM). You may wish to disable this if you have special initialization requirements. The AUX device driver makes BIOS calls (through INT 14h) to program the serial ports.

If you are using a communications (COMM) library to handle RS-232 I/O on an interrupt basis, or even if your application polls the 8250's manually, you may need to use the IRQPRIORITY= statement in CONFIG.SYS to give thread preemption (based on IRQ0) a lower priority than your RS-232 device interrupt (IRQ3 or IRQ4). If you do not do this, then IRQ0 can preempt your IRQ3/IRQ4 ISRs, and the EMBEDDED DOS 6-XL scheduler could cause your ISR to be suspended as though it were an ordinary round-robin task.

If you are polling for RS-232 activities, you may wish to avoid calling the *kbhit* C library function directly. Some versions of this function as provided by C compiler vendors block-out interrupts for extended periods, causing RS-232 interrupts to be missed. They may also make EMBEDDED DOS 6-XL function calls that are necessarily more complex than the generic DOS counterparts, adding overhead in a CPU-bound polling loop. Use *_intdos* instead to make an INT 21h call directly to have the same effect without the overhead. If needed, you can further reduce latency by turning BREAK OFF with an INT 21h function, reducing calls to the CON device driver on every INT 21h call to check for ^C.

Utility Program Issues

The disk utility programs are meant to run only under EMBEDDED DOS 6-XL. Do not use them with non-DOS disks or when running generic DOS systems. Doing so could result in file system problems.

You should not run generic DOS utilities under EMBEDDED DOS 6-XL, as EMBEDDED DOS 6-XL has different internal structures which would not match the utilities that come with another DOS.

Do not use the disk utilities with double-space drives; this can lead to file system problems.

The CHKDSK program is not compatible with doublespace (space optimized) versions of MS-DOS 5.0 and MS-DOS 6.0; do not use it to correct these file systems. Instead, boot MS-DOS and run its CHKDSK program that knows about the space optimizations used by these file systems. Alternatively, use the EMBEDDED DOS 6-XL FORMAT, SYS, and CHKDSK utilities to maintain an EMBEDDED DOS 6-XL-compatible file system.

If CHKDSK, FORMAT, DISKCOMP, DISKCOPY, SYS, or FDISK fail under EMBEDDED DOS 6-XL, verify that DISK IOCTL support has been enabled in OEM.INC for the system you are running. These utilities require this support.

Stack Overflow Issues

Some compiler vendors provide "*stack probes*" as a compile-time option for C code. This feature creates code at the beginning of each function that allocates the space for auto variables on the stack, and at the same time verifies that enough space exists for the application's stack. This check is made by comparing SS and SP against limit values that are valid for the program's initial stack, but not for stacks maintained by other threads that your application may allocate.

Because additional threads in an application run on their own stacks, the stack probe logic will fail; it is therefore important that these threads do not execute code with stack probes in their codepaths.

You can disable stack probes in your own application code with a compiler switch option. This does not stop the C library functions from having stack probes, however. It also does not guarantee that a third-party library vendor will have stack probes removed from their libraries.

Syntax Error Messages

The two command interpreters (GENERAL\UTIL\COMMAND.COM and GENERAL\UTIL\TINYCMD.COM) have slightly different syntax requirements. Be aware that the tiny one was intended to have limited functionality.

Debugger syntax error messages are usually caused by incorrectly entering hex addresses, or adding spaces or tabs between the components of a 16:16 hex address. Use xxxx:yyyy as the form for a hex 16:16 address. If either *xxxx* or *yyyy* has leading zeros, you may omit them.

Debugger syntax errors can also be caused by incorrectly specifing additional parameters that are not necessary for the command's function.

Unexpected Application Crashes

If your program crashes unexpectedly, first try the suggestions in this section, and then move on to other sections in related areas. You may need an emulator or software ICE to find the source of the crash (because these systems maintain instruction execution history), or you may need to divide-and-conquer the problem until you can isolate the specific sequence of code that causes a failure in your system. Debugging with other methods can then proceed on the specific section.

All of the EMBEDDED DOS 6-XL libraries, such as LIB\KERNEL.LIB, contain large model (far code and far data) objects. Do not call the kernel functions in a near code environment without explicitly declaring the routines as FAR. Do not pass near pointers to kernel functions without casting them FAR, or incorrect results will occur.

When expanding your application with additional threads, timers, events, mutexes, open files, and so on, make sure that your system pool is sufficiently large enough to hold all of the system data structures needed to support the objects. Use the debugger's POOL command to see the layout of system pool (including unallocated pool) and increase memory as required with the SYSTEMPOOL= statement in CONFIG.SYS.

When writing multithreaded code (code using EMBEDDED DOS 6-XL threads or timers), be sure that your code automatically sets DS to DGROUP upon entering a new thread or timer function. The scheduler does not automatically initialize DS or ES to point to your data segment.

Multithreaded applications should also be compiled with a compile-time switch that tells the compiler that it cannot make the assumption that SS==DS. By default, MSC does this, so you will need to use the /Alfu switch on the command line to fix the problem.

Calling nonreentrant C library functions, or nonreentrant third-party code with multiple threads simultaneously can cause system failures. Use EMBEDDED DOS 6-XL mutexes or critical sections to prevent this.

Calling C library functions, or third-party functions, with stack probes enabled can cause failures. Disable stack probes or use CHELPER.ASM to re-define the stack probe procedure itself, so that stack probes become benign.

Interrupt service routines that work in generic versions of DOS may not work under EMBEDDED DOS 6-XL due to increased use of the system stack during scheduling. You should always switch stacks to a private stack when entering an ISR, and switch back after you are finished processing the request.

If your threads use large amounts of run-time stack, you may need to increase their stack sizes with the STACKSIZE= parameter in CONFIG.SYS. A good number to start with is 1024 or 2048. Allocate structures on the heap with malloc or INT 21h function 48h instead of automatically on the stack to reduce stack size requirements.

Error Messages

The following is a list of error messages displayed by DOS, both COMMAND.COMs, and EMBEDDED DOS 6-XL Development Tools. Messages reported by utilities are not included, since they are common to generic DOS and are self-explaining. This list is alphabetical.

Message `[386 Optimizations Enabled]`

Origin DOS.SYS

Explanation The EMBEDDED DOS 6-XL system has detected an Intel 80386 or later CPU, capable of performing double-word memory moves to increase file system cache performance. The double-word move optimization will be used whenever possible.

Actions None.

Message `A UMB at segment` *`yyyy`* `does not exist.`

Origin DOS.SYS

Explanation The EMBEDDED DOS 6-XL system initialization module has read a UMB command that specifies the hexadecimal segment address of an upper memory

block to be linked into the memory arena, but the memory block is not properly formatted. Typically, this is because it was not properly mapped by a memory manager such as QEMM386.SYS.

Actions Verify that the memory exists by using the kernel debugger to inspect it with a 'D' (dump bytes) command. Verify that the specified address is included in the proper parameters for the memory manager used. Verify that the memory manager was loaded properly. All device drivers are loaded before UMB statements are processed, so the ordering of the DEVICE and UMB statements does not matter (the UMB statements must be ordered by segment number, however).

Message `ALL THREADS EVAPORATED, SYSTEM HALTED.`

Origin DOS.SYS

Explanation The EMBEDDED DOS 6-XL system has detected that all threads (including the idle thread) have become blocked, and the OEM has configured the stack to halt the system if such an event would occur at run time.

Ordinarily, this option is used during debugging to identify situations where it is illegal to have no runnable thread in the system. Having no runnable threads does not mean that an external hardware interrupt service routine could not allocate a new thread to run, so this is configurable by the OEM.

The only reason for the idle thread to become blocked is for an interrupt routine to make a blocking system call (such as WaitEvent or AcquireMutex). Because these task-time calls affect the currently-running task, blocking in an interrupt service routine should be considered a programming error.

Actions Verify that ISRs in the application code do not make blocking calls, or if blocking calls need to be made in response to interrupt-time activities, allocate a thread to do the blocking at task time while inside the ISR.

Message `Abort, Retry, Fail?`

Origin COMMAND.COM

Explanation A "critical error" (INT 24h) has been generated by the EMBEDDED DOS 6-XL I/O system, and the standard COMMAND.COM interpreter's critical error handler has intercepted the call. Critical errors are generated at the device driver level when drivers return a failing status code; for example, DEVERR_GENERAL_FAILURE.

Actions Respond by typing a 'A' to abort the currently running program, 'R' to retry the DOS call that caused a device driver to be called, or 'F' to cause the DOS call to return with a failing status code.

Message `Are you sure? (Y/N):`

Origin Tiny COMMAND.COM

Explanation The command, "DEL *.*" has been issued, and COMMAND.COM is verifying that you truly intend to delete all the files in that directory.

Actions Respond by typing 'Y' to delete the files, or 'N' to abort the command.

Message `Bad Address Format`

Origin DOS.SYS (Kernel Debugger)

Explanation The kernel debugger has parsed a command with an address that is malformed. The command has been rejected.

Multithreaded application programs can encounter this message if they are not compiled with the proper switches (SS!=DS, no stack probes, etc.) If system pool becomes corrupted due to a wild pointer being used in an application program, control can pass to the routine that prints this message.

Actions If you have typed a debugger command, review it and retype the corrected command. If you have run a multithreaded application, your application has been compiled incorrectly, or has incorrectly used system services, or has called nonreentrant library functions with at least two threads simultaneously. Review and correct the application's code and build procedure.

Message `BREAK is ON|OFF`

Origin Both COMMAND.COMs

Explanation The command interpreter has printed the status of the system BREAK flag in response to a BREAK command.

Actions None.

Message `DOS Version n.n`

Origin Both COMMAND.COMs

Explanation The command interpreter has printed the DOS version number (major version first, followed by a dot and then the minor version).

Actions None.

Message `DOS`*`nnnn`*`: Unable to run shell "A:\COMMAND.COM".`

Origin DOS.SYS

Explanation The system initialization sequence has attempted to execute a DOS OPEN INT 21h function to open the named command interpreter, but the open failed. This is either because COMMAND.COM was not found on the specified drive, or because the drive is a ROM disk BIOS extension that is reporting different geometry through INT 13h, function 08h, than is found in the ROM disk's BPB (in the boot record). This can happen when using the wrong floppy disk size when using the ROM disk software.

Actions Verify that the drive contains the specified file. If it does, then verify that the type of floppy used to build the ROM disk image has the same specifications as the ROM disk software expects. The boot record geometry and the ROM disk INT 13h function 08h geometry must match.

Message `DOS`*`nnnn`*`: xxxx.`

Origin Both COMMAND.COMs

Explanation The command interpreter has executed a command resulting in a system call that reported a failing status. The failing status code is represented by *nnnn* in the message, and the summary of the message code is represented by *xxxx*. Refer to DOSERR.INC for on-line versions of these status codes.

Actions Review the explanation and relate it to the problem; i.e., if "DOS0008: Insufficient memory." is displayed, then use the debugger to display the system pool with the POOL command to see if enough pool exists for another file to be opened, etc.

Message `ECHO is ON|OFF`

Origin Both COMMAND.COMs

Explanation The command interpreter has printed the status of the COMMAND.COM ECHO flag in response to an ECHO command.

Actions None.

Message `EMBEDDED DOS 6-XL, DOS Emulation Version n.n`

Origin Standard COMMAND.COM

Explanation The command interpreter has printed the version compliance number (n.n) of the underlying EMBEDDED DOS 6-XL INT 21h API in response to a VER command.

Actions None.

Message `Environment space size out of bounds`

Origin Standard COMMAND.COM

Explanation The command interpreter has detected that the /e:*nnnn* switch on the command line specifies too many bytes to allocate for the environment space.

Actions Reduce the requested environment space size.

Message `Error accessing volume in drive d:`

Origin Standard COMMAND.COM

Explanation The command interpreter requested the size of the volume in drive d: with a DOS call, and DOS was unable to return the value successfully.

Actions Insert the disk or removable disk into the drive with the drive door closed and retry the operation.

Message `Error: COMMAND.COM cannot adjust resident size`

Origin Standard COMMAND.COM

Explanation The command interpreter was unable to install its resident portion in memory. This error should not occur.

Actions Re-build the COMMAND.COM program and monitor its progress to ensure that the program was properly built.

Message *fn1* `->` *fn2*`: Copy? (Y/N):`

Origin Standard COMMAND.COM

Explanation The command interpreter received a COPY command with the confirm switch, and is asking for verification before copying file *fn1* to *fn2*.

Actions Press 'Y' to copy the file, or 'N' to skip the copy process for the given file.

Message *fn1* `->` *fn2*`: Rename? (Y/N):`

Origin Standard COMMAND.COM

Explanation The command interpreter received a RENAME command with the confirm switch, and is asking for verification before renaming file *fn1* to *fn2*.

Actions Press 'Y' to rename the file, or 'N' to skip the rename process for the given file.

Message `fn1: Delete? (Y/N):`

Origin Standard COMMAND.COM

Explanation The command interpreter received a DELETE or ERASE command with the confirm switch, and is asking for verification before deleting the file *fn1*.

Actions Press 'Y' to copy the file, or 'N' to skip the delete process for the given file.

Message `General Software EMBEDDED DOS 6-XL`

Origin Standard COMMAND.COM

Explanation The command interpreter has processed a VER command by indicating that the EMBEDDED DOS 6-XL system file is running.

Actions None.

Message `Kernel memory allocation failure.`

Origin DOS.SYS

Explanation The EMBEDDED DOS 6-XL system was unable to allocate enough memory for system pool alone.

Actions This error occurs only in systems with severely limited RAM space. Reduce the system pool requirements with the SYSTEMPOOL= statement in CONFIG.SYS, or rebuild the EMBEDDED DOS 6-XL kernel with lower minimum parameter settings in OEM.INC.

Message `Label "xxx" not found.`

Origin Tiny COMMAND.COM

Explanation The command interpreter executed a GOTO statement in a batch file, but the label *xxx* was not found in the batch file.

Actions Check the spelling of the label.

Message `LowMem/HiMem = nnnnk/nnnnk`

Origin DOS.SYS

Explanation The system initialization sequence has determined the amount of low and high memory, and has displayed it on the console. This message can be disabled with an option in OEM.INC.

Actions None.

Message `No files found.`

Origin Tiny COMMAND.COM

Explanation The command interpreter executed a DIR command, but there were no files to list.

Actions Possibly move to the proper drive or directory.

Message `One file copied.`

Origin Tiny COMMAND.COM

Explanation The command interpreter executed COPY command and copied a file.

Actions None.

Message `Out of environment space.`

Origin Both COMMAND.COMs

Explanation The command interpreter cannot complete the processing of a SET command because there is no room to store the variable in the environment space.

Actions Choose smaller names, or use the /e:*nnnn* option on the COMMAND interpreter from the SHELL= statement in CONFIG.SYS.

Message `Press any key to continue. . .`

Origin Standard COMMAND.COM

Explanation The command interpreter has processed a command that has paused its display for the user to review it.

Actions Press the spacebar or Enter to continue the listing, or ^C to terminate the listing.

Message `Press any key when ready. . .`

Origin Tiny COMMAND.COM

Explanation The command interpreter has processed a PAUSE command that has paused its display for the user to indicate that it should continue.

Actions Press the spacebar or Enter to continue the batch file, or ^C to terminate it.

Message `Ready:`

Origin Tiny COMMAND.COM

Explanation The command interpreter is waiting for you to type a command.

Actions Type a legal command for the tiny COMMAND.COM command interpreter and press Enter to have it process the command.

Message `Strike a key when ready. . .`

Origin Standard COMMAND.COM

Explanation The command interpreter has processed a PAUSE command that has paused its display for the user to indicate that it should continue.

Actions Press the spacebar or Enter to continue the batch file, or ^C to terminate it.

Message `SWITCH is 'x'.`

Origin Standard COMMAND.COM

Explanation The command interpreter has processed a SWITCH command without operands, causing it to display the current switch character.

Actions None.

Message `Syntax error.`

Origin Both COMMAND.COMs

Explanation The command interpreter has detected an invalid internal command because it was malformed, or had the wrong number of arguments, or had the wrong arguments.

Actions Review the command and correct it.

Message `System pool could not be resized from CONFIG.SYS.`

Origin DOS.SYS

Explanation The EMBEDDED DOS 6-XL system initialization processed a SYSTEMPOOL= command in CONFIG.SYS that requested less memory than the minimum pool size configured during system build time. Ordinarily, this number is around 8096-16384. The size of the pool cannot exceed 64K minus a small amount of static memory used in the 64K system area; roughly 60K should be the upper bound.

Actions Specify a system pool size above the minimum configured value and less than the maximum possible size.

Message `Terminate batch file? (Y/N):`

Origin Tiny COMMAND.COM

Explanation The command interpreter has received the operator's ^C type-in during execution of a batch file and is prompting for verification before stopping the batch file's execution.

Actions Press 'Y' to terminate the batch file and return to interactive mode, or 'N' to continue executing the next command in the file.

Message `Terminate batch process? (Y/N):`

Origin Standard COMMAND.COM

Explanation The command interpreter has received the operator's ^C type-in during execution of a batch file and is prompting for verification before stopping the batch file's execution.

Actions Press 'Y' to terminate the batch file and return to interactive mode, or 'N' to continue executing the next command in the file.

Message `The` *`xxx`* `command is not available.`

Origin Standard COMMAND.COM

Explanation The command interpreter has attempted to process an internal command that is recognized in its table; however, it has been disabled in the source. Ordinarily, OEMs may disable commands that are unnecessary to an application to save space.

Actions Enable the command in the COMMAND.COM source.

Message `The "xxx" CONFIG.SYS option must be "ON" or "OFF".`

Origin DOS.SYS

Explanation The EMBEDDED DOS 6-XL system initialization module has read a CONFIG.SYS command that requires the syntax, *xxx*=ON or *xxx*=OFF, and this syntax was violated.

Actions Correct the syntax of the CONFIG.SYS command.

Message `The "xxx" CONFIG.SYS driver could not be loaded.`

Origin DOS.SYS

Explanation The EMBEDDED DOS 6-XL system initialization module has read a CONFIG.SYS DEVICE= command that specified the name of a driver file to load into the system, but the system could not find the file. Either the file does not exist or the geometry for the device does not match the corresponding boot record on the device. This can happen when building the ROM disk image and incorrect parameters are specified to the ROM disk BIOS extension software.

Actions Verify that the device driver's name is correct, that the driver file is on the boot device, and if necessary verify that the ROM disk was built correctly. Refer to the Embedded BIOS System Integrator's Guide for details on ROM disk troubleshooting when using the Embedded BIOS ROM disks.

Message `The "xxx" CONFIG.SYS driver terminated itself.`

Origin DOS.SYS

Explanation The EMBEDDED DOS 6-XL system initialization module has loaded a device driver in response to a DEVICE= statement in CONFIG.SYS, but the device driver failed to initialize itself.

Actions Review the messages issued by the device driver and take corrective action based on the device driver manufacturer's troubleshooting instructions.

Message `The "xxx" CONFIG.SYS parameter must be set to a number.`

Origin DOS.SYS

Explanation The EMBEDDED DOS 6-XL system initialization module has read a CONFIG.SYS command that requires the syntax, *xxx=nnnnn* (where *nnnnn* is a digit string), and this syntax was violated.

Actions Correct the syntax of the CONFIG.SYS command.

Message `The "xxx" CONFIG.SYS parameter must be set to a hexadecimal number.`

Origin DOS.SYS

Explanation The EMBEDDED DOS 6-XL system initialization module has read a CONFIG.SYS command that requires the syntax, *xxx=yyyy* (where *yyyy* is a hexadecimal (0-9,A-F) digit string), and this syntax was violated.

Actions Correct the syntax of the CONFIG.SYS command.

Message `The "xxx" CONFIG.SYS parameter must be set to a number.`

Origin DOS.SYS

Explanation The EMBEDDED DOS 6-XL system initialization module has read a CONFIG.SYS command that requires the syntax, *xxx=nnnnn* (where *nnnnn* is a digit string), and this syntax was violated.

Actions Correct the syntax of the CONFIG.SYS command.

Message `The "xxx" CONFIG.SYS keyword is unrecognized.`

Origin DOS.SYS

Explanation The EMBEDDED DOS 6-XL system initialization module has read a CONFIG.SYS command that is unrecognized.

Actions Refer to the CONFIG.SYS reference chapter in this manual for a list of recognized CONFIG.SYS commands.

Message `The system could not allocate additional stacks.`

Origin DOS.SYS

Explanation The EMBEDDED DOS 6-XL system initialization module has read a CONFIG.SYS STACKS= command that would allocate more memory than is available.

Actions Reduce the number of stacks or the size of each stack with the STACKSIZE= command.

Message `The UMB=yyyy statement is out of an ascending UMB sequence.`

Origin DOS.SYS

Explanation The EMBEDDED DOS 6-XL system initialization module has read a CONFIG.SYS UMB command that requires specifying a 4-digit hexadecimal segment address. All UMB statements in the CONFIG.SYS file must be ordered in the file in ascending order of hexadecimal segment addresses.

Actions Re-order the UMB statements so that their addresses are sorted from lowest to highest addresses.

Message `This COMMAND.COM can only be run in the EMBEDDED DOS 6-XL environment.`

Origin Tiny COMMAND.COM

Explanation The command interpreter was run as a program but EMBEDDED DOS 6-XL is not running. The COMMAND.COM interpreter requires EMBEDDED DOS 6-XL to run..

Actions Boot EMBEDDED DOS 6-XL before running the command interpreter for EMBEDDED DOS 6-XL.

Message `Tiny Command Processor Vn.n`

Origin Tiny COMMAND.COM

Explanation The command interpreter has booted for the first time, or has encountered a critical error from a running program or during processing of a command.

Actions None, or identify the source of the critical error (drive door open, etc.)

Message `Usage: COMMAND [/C string] [/E:nnnn] [/P] [/S]`

Origin Standard COMMAND.COM

Explanation The command interpreter has detected an error during processing of its command line switches.

Actions Call the command interpreter with one of the listed switches.

Message `VERIFY is ON|OFF.`

Origin Standard COMMAND.COM

Explanation The command interpreter has processed a VERIFY command without operands, causing it to display the current DOS verify switch status. If VERIFY is ON, then the file system is operating in hardened mode. If VERIFY is OFF, then the write-behind cache algorithm is enabled.

Actions None.

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W