FlexOS™
System Guide

Version 1.3

1073-2013-001

First Edition: November 1986

Foreword

This guide is for the original equipment manufacturer (OEM) and system programmers who install and use FlexOS.

The FlexOS System Guide is intended for the original equipment manufacturer (OEM) and system programmer responsible for implementing FlexOS on a specific computer system. The text describes FlexOS architecture, its interface to hardware devices, and the functions available to the driver writer. To use this guide, you should be familiar with device drivers and the C programming language.

Digital Research supplies FlexOS as a set of compiled operating system modules and device drivers for a variety of consoles, disk drivers, and printers. You can compile these samples to interface to the corresponding device or use them as models for building your own hardware interfaces.

The driver routines are written in the C programming language and make use of the FlexOS C run-time library. Although FlexOS allows you to write device drivers entirely in C, you might need or prefer to use assembly language routines in your code where speed of execution cannot be compromised.

To complete your understanding of FlexOS, you should read the FlexOS Programmer's Guide for its programming interface and Supervisor calls description. For the description of the user interface, read the FlexOS User's Guide. The FlexOS documentation set also includes the supplements describing important, microprocessor-specific information. Refer to the supplement corresponding to the microprocessor in your computer.

Hardware Requirements

You can tailor FlexOS to run in systems based on a variety of microprocessors. While FlexOS can take advantage of built-in memory management found in advanced microprocessors, it does not require such memory management units to create a multitasking environment. See the processor-specific supplements to this manual for more information on memory management.

Digital Research® suggests that a minimum FlexOS system contain 512 kilobytes of RAM. FlexOS supports 2 gigabytes of disk storage space.

FlexOS supports a variety of clock devices and memory management units. FlexOS supplies application programmers with a standard console and disk interface that supports integrated multi-window and desktop applications. FlexOS provides support for a broad range of hardware environments.

Sample device driver code is distributed as models for floppy and hard disk drives; serial, bit-mapped, and character-mapped consoles; and serial I/O and printer ports.

About this Manual

This manual is organized into the following sections:

Section 1 Introduction to the FlexOS operating system.

Section 2 Overview of the FlexOS I/O system.

Section 3 System configuration.

Section 4 Synchronous driver interface to Resource Managers.

Section 5 Use and function of the FlexOS driver services.

Section 6 Driver interface to the Supervisor module.

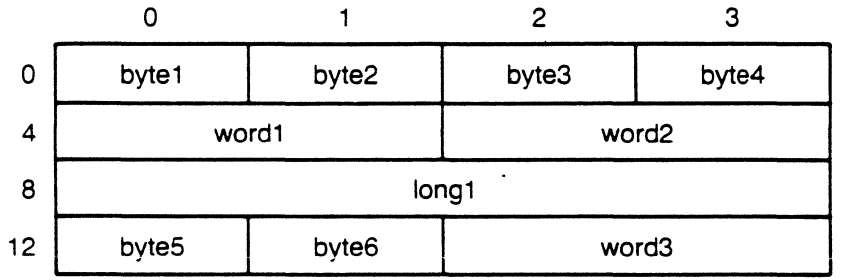
Sections 7 through **11**
I/O functions for console, disk, printer, port, and special drivers.

Section 12 The FlexOS bootstrap program requirements, memory image, and SYS utility.

Appendixes System character sets, support for foreign languages, and window management.

Data Structure Convention

Throughout this manual, data structures are represented in diagram form as shown below. The corresponding C listing for the diagram follows the illustration. Word and byte order are important when using these structures.



Data Structure

```
struct thisstruct
{
        BYTE    byte1;  /* byte offset = 0  */
        BYTE    byte2;  /* byte offset = 1  */
        BYTE    byte3;  /* byte offset = 2  */
        BYTE    byte4;  /* byte offset = 3  */
        WORD    word1;  /* byte offset = 4  */
        WORD    word2;  /* byte offset = 6  */
        LONG    long1;  /* byte offset = 8  */
        BYTE    byte5;  /* byte offset = 12 */
        BYTE    byte6;  /* byte offset = 13 */
        WORD    word3;  /* byte offset = 14 */
};                      /*       length = 16 */
```

Contents

Tables

Figures

Listings

System Overview

This section presents a basic overview of the FlexOS operating system, including a description of its various modules.

1.1 Features

FlexOS is a real-time, multitasking operating system for single- and multi-user microcomputer systems. It is written to be independent of a system's microprocessor and peripheral equipment. FlexOS's programming interface allows a programmer to take maximum advantage of advanced hardware technology, such as bit-mapped graphics devices or high-capacity disk storage units. The programming interface is machine-independent, so applications need not be rewritten for different machines or different sets of peripherals.

The following is a list of the prominent features of FlexOS:

- Runs multiple applications in an asynchronous environment that allows real-time response to external events.
- Interfaces to microprocessors that provide memory mapping and memory protection through memory management hardware.
- Allows inter-process communication synchronization through a pipe system.
- Allows asynchronous I/O and timing through an event system. A process can wait for multiple events or handle asynchronous events through software interrupts.
- Provides a standard terminal interface and standard character- and bit-mapped screen interfaces.

- Provides standard keyboard interfaces independent of physical console types. Supports 8-bit and 16-bit keyboard input modes with keyboard translation, which allows support for special characters, function keys, multi-keyed characters and foreign languages with 16-bit characters, such as KANJI.
- Manages multiple virtual consoles on each physical console.
- Supports real-time data acquisition and background communications.
- Allows device drivers to be linked with the system or dynamically loaded at run-time.
- Supplies country codes to determine character set, accounting, monetary, and date presentation.
- Provides applications with full error recovery facilities from physical I/O errors on any device.

1.2 Operating System Organization

Figure 1-1 illustrates three distinct parts of FlexOS.

The FlexOS **program** part consists of utilities, user shell programs, shared run-time libraries, applications, window managers, and any other loadable programs calling operating system services. You define the FlexOS user interface in the program portion.

The **system** part contains the device and program independent portions of the operating system. The elements of this portion are the Supervisor, kernel and resource manager.

The **physical** part contains all of the system's device-dependent code for the system's disk drives, consoles, and other peripheral devices. The code is organized in the form of independent drivers controlled by a single resource manager.

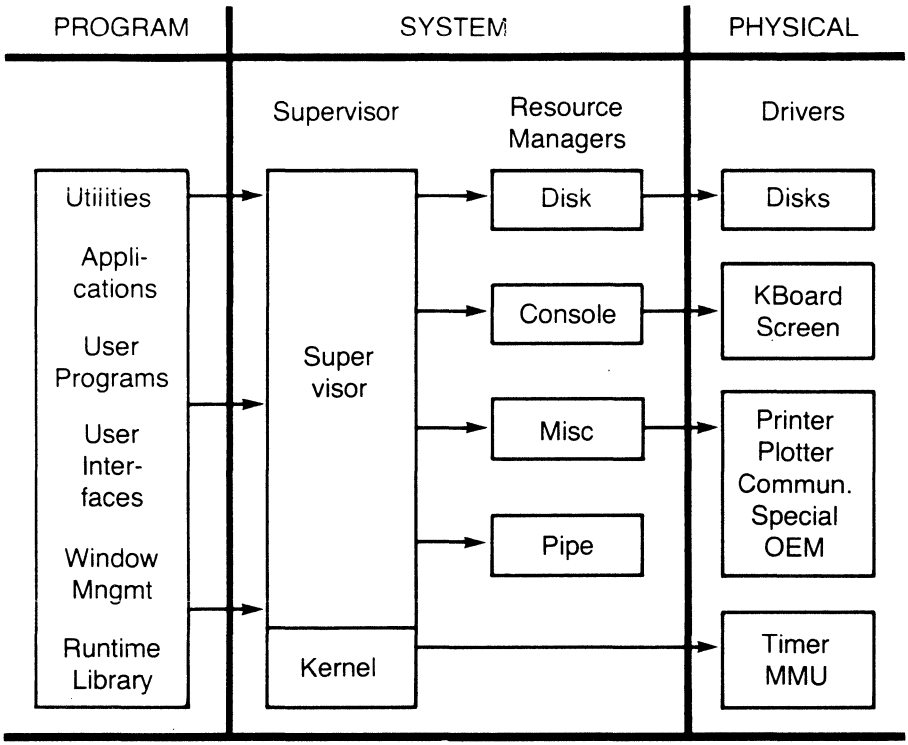


Figure 1-1. Structure of FlexOS Operating System

1.2.1 Programs

All programs loaded from disk, whether applications, utilities, or user interface programs, run independent of each other. Each runs under a process context and has separately addressable User Memory. Programs call the Supervisor for operating system or machine services.

1.2.2 The Supervisor Module

The Supervisor controls the flow of requests to the Resource Managers. The Supervisor parses all names and file numbers to determine which Resource Manager should obtain a function request. It handles the DEFINE, CONTROL, COMMAND and OVERLAY Supervisor Calls (SVCs) directly. These functions do no actual resource management, but call the Resource Managers for services.

1.2.3 The Kernel

The kernel manages processes and memory. This facilitates intermachine communications, networking, multi-user, and multitasking environments. The kernel controls the timer driver and any special routines for memory management based on the type of memory management unit. The kernel is based on an event-driven dispatcher that schedules on a priority basis. Time slicing is done by a timer event occuring once per tick. A tick occurs every 16 to 20 milliseconds, depending on the implementation of the Timer driver. Scheduling of equal priority processes is done in a round-robin fashion.

1.2.4 Resource Managers

A Resource Manager controls the resources associated with it and provides a standard interface to the device drivers for each category of device.

The Disk Resource Manager manages the disk file systems on disk drives. It supports DOS disk media and provides a single interface to floppy and hard disks.

The Console Resource Manager manages physical consoles, including the screen and keyboard devices. FlexOS supports virtual consoles through the console drivers. Applications create virtual consoles through a Supervisor Call.

The Miscellaneous Resource Manager manages all devices not managed by other Resource Managers, including plotters, printers, ports, and communications devices. Drivers for these devices are referred to as special drivers. The Miscellaneous Resource Manager places more responsibility on the driver than the other Resource

Managers do, passing requests from a calling process to the appropriate special driver with a minimum of processing.

The Pipe Resource Manager manages interprocess communications and synchronization through named FIFO memory files called pipes. These "in-memory" files are used to pass messages from one process to another or to synchronize activities. There are no devices associated with the Pipe Resource Manager.

1.2.5 Device Drivers

A device driver is the software interface to a physical device. Device drivers contain all of the machine- and device-specific code in the system. Each driver is separately built and is independent of other drivers.

A device driver is managed by a single resource manager. A device driver can, however, control multiple devices of the same type. For example, a disk driver can control a disk controller, which in turn controls multiple disk drives. Table 1-1 lists the driver types and indicates the resource manager that controls it.

Table 1-1. Driver/Resource Manager Relationships

Driver Type	Resource Manager
Disk	Disk
Console	Console
Port	Miscellaneous
Printer	Miscellaneous
Communications	Miscellaneous
Special OEM	Miscellaneous

There are three ways to install drivers:

- Integrating the drivers with the system.
- Installing them dynamically when FlexOS is loaded.

- Installing them when the system is up and running.

Link the compiled driver files into the system image to permanently install a driver. To install a driver while FlexOS is loading, add a DVRLOAD command to the boot script. Use the DVRLOAD command once the system is running or add an INSTALL supervisor call to your application to install a driver once the system is up and running.

FlexOS also supports sub-drivers. A sub-driver is constructed just like a driver, however, the sub-driver is controlled by the driver rather than a resource manager. In this case, the controlling driver functions as the subordinate driver's resource manager. Through sub-drivers, one driver can control multiple devices of the same type with different I/O interfaces.

1.3 File Management

FlexOS has a hierarchical, shared-disk file system with record- and file-locking mechanisms. The disk file system is protected at several levels. Access to files is based on file and directory ownership through user and group identification numbers. Users identify themselves through login procedures that can include password protection. The disk file system thus provides integrity and data protection in both multi-user and single-user systems.

FlexOS distinguishes between removable and permanent media. It gives special recognition to removable media on devices supporting open door interrupts. By knowing the environment, FlexOS optimizes performance and minimizes lost data.

1.4 Memory Management

FlexOS supports mapped and protected memory management hardware. Because of the diversity in memory management units, FlexOS supports a simple memory model that maps into the more common MMUs on various CPUs.

1.5 Printer Management: Print Spooler

FlexOS includes a print spooler system in the form of a driver. To include the spooler, load the driver SPLDVR.DVR, define a subdirectory as tempdir: (the subdirectory must exist), and make a couple of logical name definitions for the system printer in your boot script. See the example boot script in Section 3 for the commands used to load the spooler driver and to make the appropriate logical name assignments.

To run the spooler, the following files must be present with your printer driver on the boot: drive:

SPLDRV.DRV	Application program interface for spooler
SPOOL	Executable spooler module
DESPOOL	Executable despooler module

In addition, you need the PRINT utility on the system: drive. Only the PRINT utility is invoked by the user. SPOOL and DESPOOL are invoked by the spooler driver.

The following spooler description assumes that the spooler driver has been installed and defined as the prn: device. If you do not intend to use the spooler, be sure to define another list device as prn:.

Note: Although you define one printer to be the spooler's output device, the spooler can make use of multiple printers. The destination printer is selected by the user via the PRINT utility. If you are going to have multiple printers, the device driver must be loaded before the selection is made. The spooler automatically links to the driver when the user requests the device; however, it cannot load the device. If the device specified is not present, the spooler returns an error message.

The spooling system has three components: the spooler driver, the spooler process, and the despooler process. The spooler driver creates the spooler and despooler processes and a set of pipes used to control the system. The driver provides both command-line and application interfaces.

The command line interface uses the PRINT utility to print files. When the user invokes PRINT, the utility parses the command line and groups the file specifications in a job. PRINT then sends each file in the job to the prn: device. Because the spooler driver is defined as the prn: device, the files are added to the spooler system's print queue.

See the FlexOS User's Guide for the description of the PRINT utility and its options.

Application programs access the spooler driver through the prn: device. Like any device, the program must open prn: before it can use it. When the application writes to the prn:, the spooler automatically creates a file in the system temporary file directory tempdir: and records the output therein. The spooler driver closes the file when the program closes the prn: and adds the file name to its print queue.

The spooler is a user process created by the spooler driver that waits on a pipe for the file names of files to be printed. The spooler driver provides the file specifications in this pipe. When a name is received, the spooler adds it to the end of a print queue. The print queue is recorded on disk in the system: directory and maintained on a first in first out. When the spooler is created, it looks to this file to see if there are any entries. Thus, if the system crashes, jobs in the queue are preserved and printed when the system is restarted.

The despooler is also a user process created by the spooler driver. The despooler reads the queue file and prints the file at the top of the list on the device assigned as the bgprn: device. When the file output is complete, the despooler removes the entry, moves the next file to the top of the queue, and prints it. If no file is present, the despooler waits for an entry to be made.

For the description of spooler use with FlexNet™, see the FlexNet User's Guide.

End of Section 1

I/O Overview

This section explains, in broad terms, how FlexOS performs input and output. It defines the system I/O modules and describes their interaction. This section also contains an overview of the flow of control in an I/O operation and concludes with a discussion of driver installation.

2.1 File-Oriented Input and Output

FlexOS performs input and output by treating a device as a special kind of file. Programs initialize I/O with the OPEN or CREATE Supervisor Call (SVC). Multiple devices and files can be open simultaneously. Use the OPEN SVC to open an existing disk file, pipe, console, or device. Use the CREATE SVC to create and open a new disk file, virtual console, or pipe.

Both OPEN and CREATE calls return a 32-bit value called the file number. This value uniquely identifies a specific communication channel between a process and a device. All device-related I/O SVCs reference this number. The Supervisor then uses the file number to decide which Resource Manager should receive the request.

You break a communication channel with the CLOSE SVC. Subsequent attempts to access the device or file return with an error message. Before the file is closed, FlexOS flushes the write buffers, unlocks locked regions, and completes outstanding asynchronous events.

The FlexOS file-oriented I/O scheme is normally device-independent. SPECIAL functions are available to perform certain device-dependent functions.

2.2 Organization of I/O Modules

This section discusses the principal components in FlexOS relating to I/O: device drivers, units, and Resource Managers.

2.2.1 Device Drivers

A device driver is the system software that translates logical I/O requests into physical commands to specific devices. Drivers contain all of the device-specific code in the system. FlexOS does not create a process for a driver. A driver's code is run by the application and system processes working through a driver's resource manager.

A driver is composed of a code group and a data group. The code group consists of a set of primitive functions that control the driver's devices. FlexOS prescribes a set of functions for each type of driver. Each driver type is described in a separate section of this manual. A driver's functions can use the SVCs available through the programmer's interface. In addition to the SVCs, drivers can take advantage of the FlexOS set of driver services, described in Section 5.

The data group contains a data structure called the Driver Header and the remainder of the driver's image. FlexOS controls a driver through its Driver Header, which contains entry points to the driver's functions, indicates whether and to what degree a driver can operate asynchronously, and holds other information about the driver. The Driver Header must be the first structure within the driver's data area.

Functions within a driver's code group access the FlexOS driver services through the Driver Services Table. FlexOS places the address of the Driver Services Table in the Driver Header.

FlexOS supports both statically- and dynamically-loadable device drivers. Both types of drivers have the same structure, so that drivers can be written without regard for the time they are linked or loaded.

2.2.2 Units

Each loaded device driver supports one or more units. Units are specific instances of physical devices. FlexOS manages devices at the unit level. Each unit is treated as an independent functional entity with its own

- name,
- access level, and
- (optionally) associated sub-driver(s).

Defined from a unit point-of-view, a device driver is a collection of functions that control related units. The only time FlexOS deals with device drivers, rather than units, is when a driver is installed or removed from the system. Both installation and removal of drivers is performed with the FlexOS INSTALL SVC command.

The organization of units into device drivers allows a higher level of management for groups of associated devices. For example, consider a system where a single disk controller controls multiple disk drives. A single device driver manages the controller, while the units designate individual drives. As far as possible, the device driver uses the same code to control all of the drives.

FlexOS logically calls each disk drive unit independent of other disk drives. However, because of driver organization, the disk device driver can force access to the individual drives to be serial.

2.2.3 Resource Managers

Each driver unit in the system is controlled by a Resource Manager. Resource Managers translate I/O requests, such as READ or WRITE, into calls to the appropriate driver unit. Typically, an application process runs the code in a Resource Manager.

Through the Driver Header, the driver makes available to the Research Manager the addresses of all its primitive functions. After receiving a function request, a Resource Manager maps the request to a specific unit and passes control to the appropriate function in the unit's driver.

A driver is defined by the type of Resource Manager controlling it. Resource Managers and their drivers fall into four categories: console, disk, kernel, and miscellaneous. FlexOS also has a Pipe Resource Manager; however the Pipe RM does not have any drivers associated with it.

2.3 Driver Unit Flow of Control

The following diagram illustrates the flow of control from an application to a driver unit.

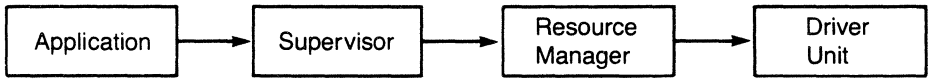


Figure 2-1. I/O Flow of Control

In Figure 2-1, a file number is passed from the application to the Supervisor, which enables the Supervisor to select the appropriate Resource Manager. In turn, information passed from the Supervisor enables the Resource Manager to select the correct unit.

2.4 Steps in Servicing I/O Request

Applications must open a file by name before they can access that file. In FlexOS, by definition, a file specification includes the driver unit name. Thus, a minimum file specification takes the form "device:filename." When the file is recorded in a subdirectory rather than the device's root directory, place the path specification between the device name and the file name.

The FlexOS Supervisor associates each device with a unique name defined through the INSTALL SVC. A device name indicates a particular driver unit and the resource manager that controls it.

When a device open call is received, the Supervisor calls the appropriate resource manager to establish the connection between the calling process and unit. If the call is successful, the Supervisor sets up internal control information and returns a file number for the opened file.

For all file I/O calls, the Supervisor translates the specified file number, calls the appropriate Resource Manager, and provides it with the control information. The Resource Manager uses this information to select the appropriate driver unit. The control information is maintained by the Supervisor until it receives a CLOSE call.

2.5 Asynchronous I/O

FlexOS supports asynchronous I/O functions in its programming and driver interfaces. Applications can start an I/O operation, perform other activities, then wait for the I/O operation to finish at a later time.

2.5.1 Support for Handling Asynchronous Events

Typically, applications use the WAIT SVC to wait for the completion of one or more I/O event, then call the RETURN SVC to obtain the return code of the completed event. To enable applications to use the FlexOS asynchronous capabilities, drivers perform their I/O asynchronously.

FlexOS provides support for asynchronous drivers with a flag system whose functions, a subset of the driver services, communicate with the WAIT and RETURN SVCs. The flag system driver services are typically called from Asynchronous Service Routines (ASRs), which in turn, are initiated by Interrupt Service Routines (ISRs). Driver writers must write their own ASRs and ISRs, according to the requirements of their hardware and the guidelines given in Section 5.

FlexOS also provides a polling function for non-interrupt-driven drivers and functions to create and declare critical regions, such as mutual exclusion regions. These mechanisms for dealing with asynchronous events are also described in Section 5.

2.5.2 Synchronous and Asynchronous Interfaces

A driver's external interfaces can be divided into two classes, synchronous and asynchronous. The synchronous interface is the interface between a resource manager and a driver's primitive functions. Processes request device I/O by calling the synchronous portion of a driver through the resource manager. The asynchronous interface is the driver's interface to the physical device, represented by a unit. The following figure illustrates the interaction between the synchronous and asynchronous portions of an I/O request.

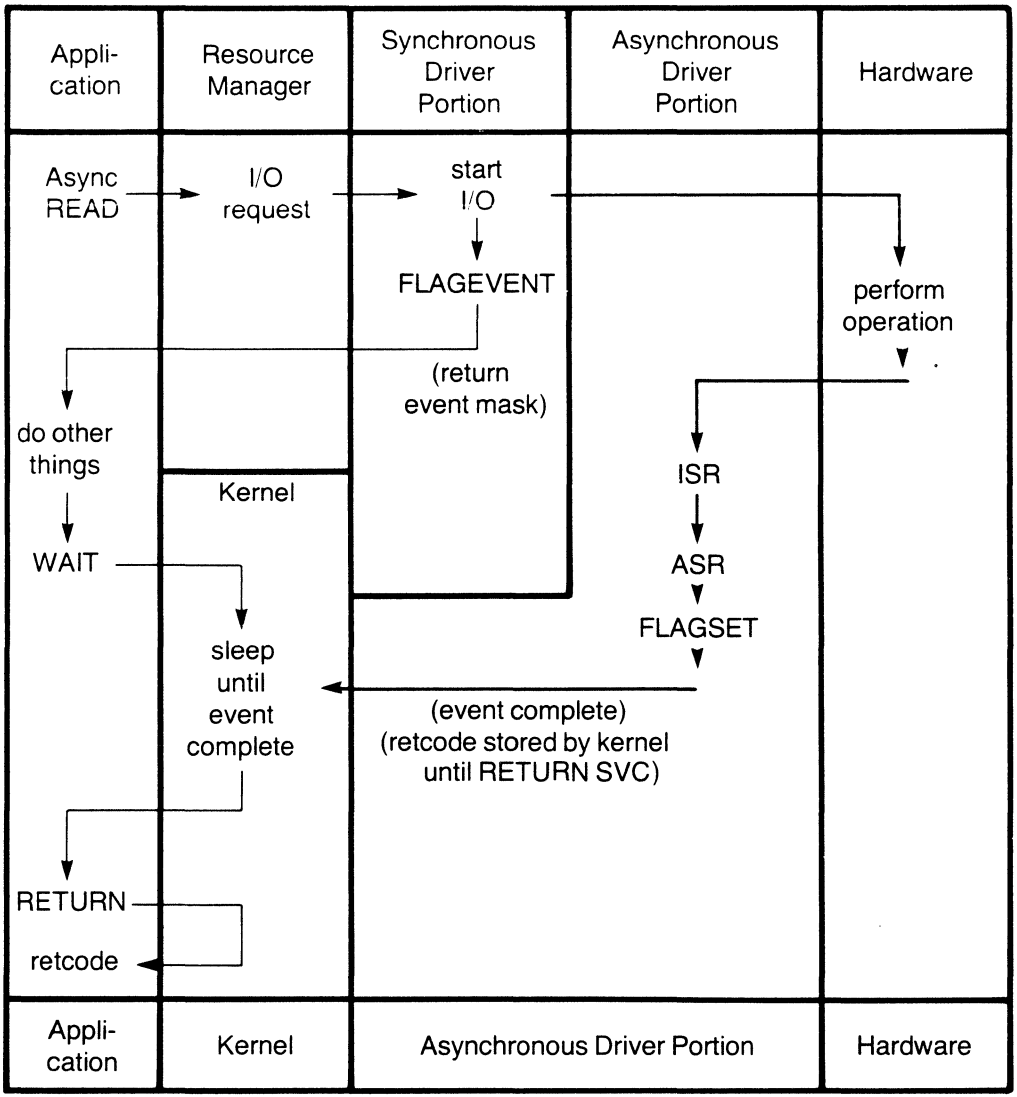


Figure 2-2. Asynchronous I/O Request

In Figure 2-2, FLAGEVENT and FLAGSET are driver services provided by FlexOS. They are described in Section 5.1.

2.6 Sub-drivers

FlexOS allows a driver to become the Resource Manager of another driver, through a concept called sub-drivers. Under this concept, a driver can access a specific piece of hardware through the functions contained in another driver, which becomes the sub-driver to the first driver. This allows the first driver to access a specific piece of hardware, while maintaining device-independence.

In response to a request from a driver, a sub-driver can respond in any of three ways. It can

- perform the requested function,
- pass the request on to its own sub-driver, or
- perform part of the function and pass the request on to a sub-driver for further processing.

Sub-drivers work at the unit level of a driver. Each driver unit can independently request one or more sub-drivers of specified types. Sub-drivers themselves are driver units.

FlexOS guarantees that each driver in the system, including sub-drivers, has only one owner at a time. The owner is either a Resource Manager or another driver.

A sub-driver's place in the FlexOS I/O scheme is shown in Figure 2-3.

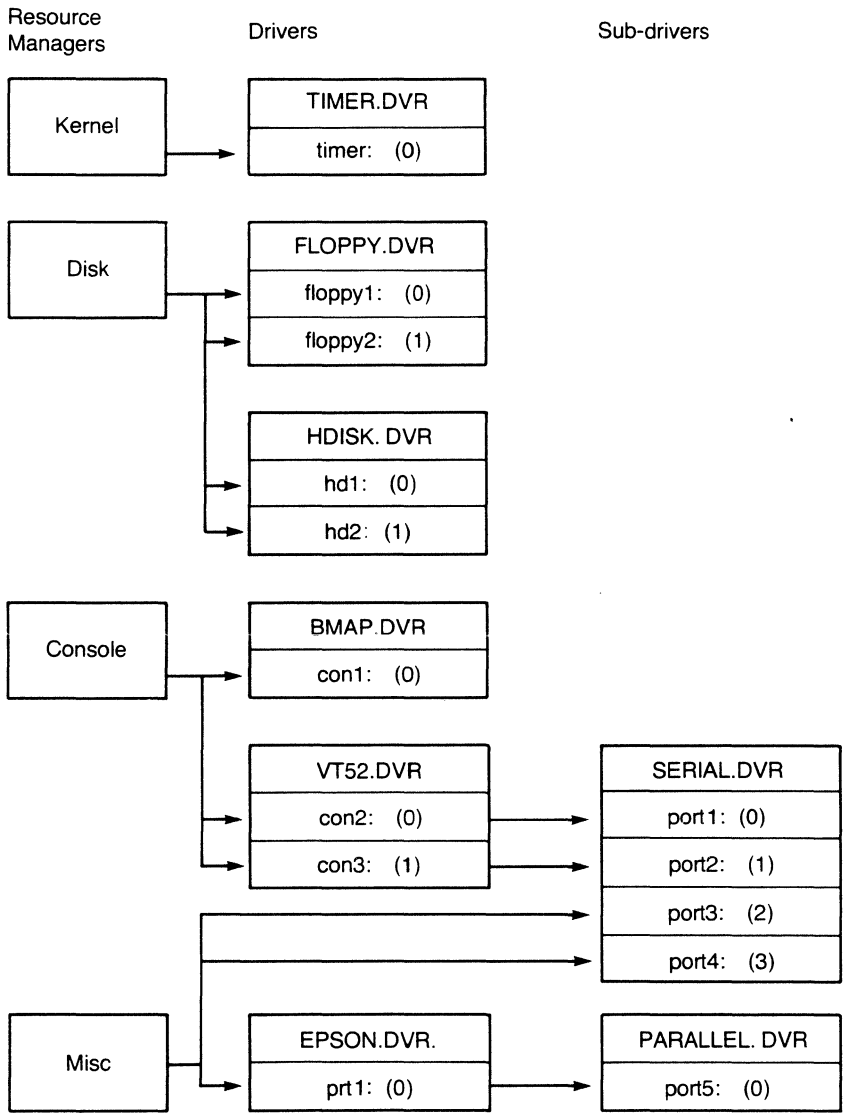


Figure 2-3. Relationship of Sub-drivers to Drivers

In the preceding figure, numbers in parentheses designate driver units. Port 1: and port 2: under the serial port driver are sub-drivers to console 2: and 3: under the VT52 driver. In this scheme, con2: and con3: are the resource managers to port1: and port2:, respectively.

At the same time, port3: and port4: under the serial port driver are drivers controlled by the Miscellaneous Resource Manager. port5: under the parallel port driver is a sub-driver to port1: of the printer driver.

Use of sub-drivers adds flexibility to a system, as the following two examples illustrate.

Through the use of sub-drivers, you can offer a system that allows the use of a number of different terminals. FlexOS lets you write a console driver for a DEC VT-100 terminal that requires a port type of sub-driver. The same VT-100 driver could drive a number of different terminals, as long as there existed port drivers to interface to the terminals' serial controllers.

To change a terminal, a user need only change a terminal emulation module, rather than replacing the entire terminal driver code. An OEM can provide several different terminal emulation modules, which a user could install through a FlexOS-supported utility or a simple boot script.

2.7 Installing Drivers

FlexOS lets you install a driver by either of two methods:

- Static: Linking a driver into the system
- Dynamic: Linking a driver to a Driver Run-time Library and loading the driver independently

Dynamically-installed drivers are loaded from disk either through a boot script or a user commands.

Drivers installed at system link time are linked into the operating system image and are loaded with FlexOS at boot time. These drivers are linked with the Driver Run-time Library, which contains addresses that the driver will need for successful operation. Dynamically-installed drivers are discussed in Section 3.

FlexOS loads drivers into memory in the same way it loads applications, with the exceptions that a driver is loaded into system space and that FlexOS does not create a process for a driver.

Because all drivers are identical in structure, drivers are written without regard for the time or method they are to be installed.

End of Section 2

System Configuration

This section explains, in general terms, how to install drivers and other implementation-specific modules in a FlexOS system. FlexOS is shipped with operating system modules, drivers, and boot loaders for target systems. If your system matches one of the target systems, you can compile, link, and load FlexOS without writing any code.

Refer to the microprocessor-specific supplements shipped with this manual for configuration information pertinent to your particular system. The System Release Notes contain specific directions for a implementing a FlexOS system based on a given CPU. They also identify the console, disk, port, and printer hardware for which FlexOS provides sample drivers.

You can use the sample drivers without modification if your system uses devices identical to those for which these drivers were written. The rest of this manual provides guidelines for writing your own drivers or modifying the sample drivers.

Section 3.1 outlines the steps involved in creating a FlexOS system. Creating a system involves linking drivers and any OEM-supplied modules into a system.

Section 3.2 describes how to edit the source code for the CONFIG module to link drivers and OEM-supplied modules into the system. The CONFIG module drives the FlexOS initialization and configuration.

Section 3.3 explains how to install drivers with the system using the boot script.

Section 3.4 explains run-time installation.

3.1 System Creation

The FlexOS OEM distribution diskettes contain the FlexOS object module files. The modules you can modify are distributed in source code form. These files have a .C file extension. The diskettes also contain all the programming tools required to compile, link, and debug a system. You create a FlexOS system using the linker provided to link the FlexOS modules and your driver modules.

3.1.1 Required Modules

The link input files (.INP files) on the FlexOS distribution diskettes indicate the various combinations of object and library files needed to create FlexOS. See the System Release Notes for detailed instructions on object link order. Typically, you link at least one disk driver into the system image.

3.1.2 Steps in Creating FlexOS

The Programmer's Utilities Guides contains explicit instructions on the use of the tools used to create a system. In general terms, the steps in creating a FlexOS system are:

1. Write FlexOS drivers and sub-drivers according to the guidelines presented in this manual or modify the sample drivers provided to match your system configuration.
2. Compile all source code with the appropriate C compiler, using appropriate options and parameters.
3. Add the names of any OEM-supplied object modules to the list of modules in the CONFIG modules. OEM-supplied modules can include user interface programs as well as drivers. Section 3.2 tells you how to modify the CONFIG modules.
4. Link the object modules with the link utility appropriate to your system's microprocessor.
5. Process the file containing the operating system file with a chip-specific FIX utility that creates a file containing the absolute memory image of FlexOS.

The boot loader gets its addresses for loading the FlexOS segments from the file produced by the FIX utility. Consequently, you can boot FlexOS in a target system without modifying the boot loader.

Section 12 explains how to build a boot disk.

3.2 The CONFIG Module

The CONFIG modules drive the configuration and initialization of FlexOS. Edit the source files of the CONFIG modules to add or delete modules for your particular FlexOS system.

Within the CONFIG modules, the list of Resource Manager modules to be linked in the operating system is contained in the MODULES Table. To add a module, you add the file specification of that module to the MODULES Table.

The code contained in the CONFIG modules is the first code run in the system at boot time. This code initializes the system modules by calling the main () routine. After the modules are initialized, the BOOTINIT function is entered.

The BOOTINIT function executes the commands in the boot script CONFIG.BAT, a modified batch file. BOOTINIT accepts standard batch commands (see the FlexOS User's Guide), the boot script commands described below, and OEM-written commands. The boot script commands call the INSTALL SVC to install drivers.

3.3 Boot Script Installation

The boot script lets you install drivers and sub-drivers, as well as user interface or window management programs, at boot time.

Drivers installed with the boot script are read from a disk file and loaded into memory. These drivers are linked with the Driver Run-time Library (DRTL), which supplies critical information on driver service routine and data addresses in a driver's Driver Header. The Driver Header is defined in Section 4.2.

A sample boot script is supplied with FlexOS in the CONFIG.BAT file. Another example is provided at the end of this section. You can

modify CONFIG.BAT or create a new boot script. You can also change the name of CONFIG.BAT. If you change the name of CONFIG.BAT, you must edit the CONFIG modules from which the boot script is called, and replace CONFIG.BAT with the new name.

3.3.1 Boot Script Commands

The boot script commands are described below. For each command, you must specify an access level. FlexOS returns an error if the access level is missing.

DVRLOAD--Load a device driver from disk

Syntax: **DVRLOAD** loadfile devicename accesslevel

Return code:
0 Success
1 Parameter error
2 Failure
3 Sub-driver needed

Where:

loadfile is the name of a loadable driver file.

devicename is the logical device name of a driver's unit 0. A colon after the devicename is optional

accesslevel is any combination of the following options:

P = Permanent driver (cannot be removed)
R = Raw READ access allowed
W = Raw WRITE access allowed
S = Raw SET access allowed
E = No exclusive access
L = Lockable (through the DEVLOCK SVC)
M = Multiple partitions allowed
N = Shared access allowed
V = Verify writes allowed

DVRUNIT--Add a new unit to an existing driver

Syntax: **DVRUNIT** olddevice devicename accesslevel

Return code:
0 Success
1 Parameter error
2 Failure
3 Sub-driver needed

Where:

olddevice is the device name of the existing driver

devicename is the logical name of the driver's new unit. A colon after the device name is optional.

accesslevel is any combination of the above options.

DVRLINK--Link an existing driver to another driver

Syntax: **DVRLINK** devicename subdriver

Return code:
0 Success
1 Parameter error
2 Failure
3 Sub-driver needed

Where:

devicename is the name of previously installed device

subdriver is the name of previously installed device which will be used as a sub-driver by the device designated by devicename.

DVRUNLK--Remove a driver

Syntax: **DVRUNLK** device:

Return code: 0 Success
2 Failure

Where:

device is the name of an installed device

Note: If the driver removed has a sub-driver associated with it, the sub-driver becomes associated with the uninstalled driver's resource manager.

3.3.2 Logical Name Definitions

A typical boot script contains logical name definitions appropriate to the hardware implementation. These definitions are made through the DEFINE SVC which updates the SYSDEF Table, which contains system-wide logical name definitions, or the PROCDEF Table, which contains definitions for a given process. Which table to modify is indicated in bit 0 of DEFINE's flags field.

The following logical names are reserved by FlexOS and should be defined in your CONFIG.BAT file:

- "system:" indicates the global system directory. It is defined in the SYSDEF Table.
- "a:" - "p:" represent the root directories of the disk drives present. For example, if there are four disk drives, they would be called drives "a:" through "d:" and drives "e:" through "p:" would be undefined. Disk drives are defined in the SYSDEF Table.

- "protect" enables or disables system-wide password protection at log on. Put the statement:

  define -s protect=on

  in CONFIG.BAT to select password protection. If password protection is not required, include the statement:

  define -s protect=off

  instead. When password protection is required, the LOGON utility prompts the user to enter his or her password. The "protect" status is defined in the PROCDEF Table.

- "shell" represents the default user interface program. The corresponding shell program is run on each virtual console and is defined in the PROCDEF Table.

- "home:" indicateas the user's initial default directory. It is defined in the PROCDEF Table.

- "default:" indicates the current directory. It is defined in the PROCDEF Table.

- "wmanager" indicates the default window manager to run on each physical console. If you don't want window management, set "wmanager" to "shell". wmanager is defined in the PROCDEF Table.

- "con:" determines the physical console that the next LOGON program runs on. The value following "con:" is changed for each invocation of LOGON. "con:" is defined in the PROCDEF Table.

- "prn:" is the logical device name for the default list device. It is defined in the SYSDEF Table

- "tempdir:" indicates the directory for temporary files. It is defined in the SYSDEF Table.

- "bgprn:" is the despooler's default output device when the user does not specify a device name with the PRINT command. If you install the spooler driver, be sure to define prn: as the bgprn: device.

The sample CONFIG.BAT defines the shell to the FlexOS "command" utility and a: as the home: and default: devices.

3.4 Run-time Driver Installation

Drivers installed at run-time are installed identically to those drivers installed via a boot script. Both kinds of installation use the INSTALL SVC. Like drivers installed with a boot script, drivers installed at run-time are linked with the Driver Run-time Library, which places critical driver service routine and process data addresses in the Driver Header. See Section 4.2 for a description of the Driver Header.

3.5 Example Boot Script

The following is an example boot script. Do not take it as a rigid template, but rather as an example showing the general mechanisms available. The CONFIG.BAT file distributed with FlexOS contains a "bare bones" boot script that you can modify according to your needs.

This example assumes that the system boots from a diskette and that the driver contained in FLOPPY.DVR was installed at system link time. The example also assumes that "floppy1:" was established at system link time as the logical device name for the boot drive.

The boot script contains a series of logical name definitions. These definitions, made through the DEFINE SVC, are explained following the listing.

Listing 3-1. Example Boot Script

```
REM  START OF BOOT SCRIPT
REM
     define switchar = -
REM
REM  Set up user's default environment.
REM  For protect=off systems, add defines for home:,
REM  wmanager:, and shell:
REM
     define -s boot:=hd1:
     define -s system:=hd0:commands/
     define -s protect=on
     security -O=RWED -G=RWED -W=RE
     define -s helplvl = 2
     define default = d:
REM
REM  Install other disk devices: floppy0: is name
REM  of floppy disk driver linked in with system
REM
     dvrunit floppy0:  floppy1: prwsln
     dvrload hd0:  floppy0:hdisk.dvr lnwrsm
     dvrunit hd0:  hd1: lnwrsm
REM
REM  Define directory on system disk for tempdir:
REM
     define -s tempdir:=system:/temp/
REM
REM  Set up the logical names A: - D: for installed
REM  drives
REM
     define -s a:=floppy0:
     define -s b:=floppy1:
     define -s c:=hd0:
     define -s d:=hd1:
REM
REM  Install serial and parallel port drivers
REM
     dvrload port1: boot:serial.dvr prwsel
     dvrunit port1: port2: prwsel
     dvrunit port1: port3: prwsel
     dvrunit port1: port4: prwsel
     dvrload port5: boot:parallel.dvr prwsel
REM
REM  Install consoles: For consoles or any driver
REM  that needs a sub-driver, you must check the
REM  error level of the driver installed.
```

```
REM
      dvrload con1: boot:bmap.dvr prwsl
      dvrload con2: boot:vt52.dvr prwsl
      if errorlevel 3 dvrlink con2: port3;
      dvrunit con2: con3: p
      if errorlevel 3 dvrlink con3: port4;
REM
REM   Note:  port1: and port2: can be accessed
REM   directly by the application program as serial
REM   ports or linked later to a dynamically
REM   installed special driver.
REM
REM   Install print spooler
REM
      dvrload prt1: boot:printer.dvr lnrws
      define -s bgprn:=prt1:
      dvrload spldrv: boot:spldvr.dvr lnrws
      define -s prn:=spldvr:
REM
REM   Startup LOGON program on consoles: LOGON opens
REM   "con:" for its physical console and runs
REM   defined wmanager.  LOGON must run in
REM   background for bootinit to continue.
REM
      define con:=con1:
      back logon
REM
      define con:=con2:
      back logon
REM
      define con:=con3:
      back logon
REM
      end
```

End of Section 3

Driver Interface

This section describes the FlexOS driver interface. The interface is discussed in terms of driver load format (4.1); the driver header, its data fields, driver type values, and interface flags (4.2); the calling conventions for interfacing with the driver I/O function entry points (4.3); and the driver installation functions (4.4).

4.1 Driver Load Format

Drivers are divided into two separate portions, the code group and the data group. The code group portion of a driver contains all of the driver's executable code. Once the driver has been loaded into memory, its code group cannot be modified. The data group contains the remainder of the driver's image including the Driver Header, the GET/SET Table, and any fixed heap areas. See Figure 4-1.

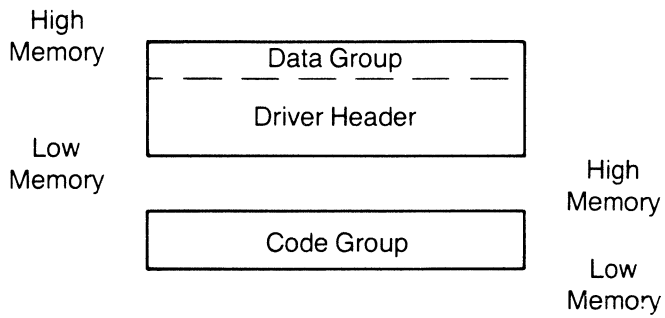


Figure 4-1. Driver Load Format

The method of loading a driver with respect to memory independence for load image portions and address relocation is dependent upon computer's CPU and the program load format. Generally, driver loading procedures are the same as those used to load normal application programs. There are three major exceptions:

- Drivers are loaded into System Space.
- A process is not created to run the driver code.
- A driver header is required in the beginning of the data group.

4.2 Driver Header

FlexOS installs and manages a driver through its driver header. The Driver Header must be at offset 0 relative to the driver's data group. It contains the entry points to the driver's functions used by the resource manager. The interface to the Driver Header entry points makes up the synchronous driver interface. Figure 4-2 shows the format of the driver header. Listing 4-1 contains a C language definition of the driver header structure. Table 4-1 describes the driver header contents.

Listing 4-1. C Language Definition of a Driver Header

```
Define struct DriverHdr
{
 UWORD  dh_reserve    ;/* Reserved                              */
 UBYTE  dh_nbrunits   ;/* Max Number of Units Supported         */
 UBYTE  dh_flags      ;/* Flag Word                             */
 LONG   dh_init()     ;/* INIT Code Entry Point                 */
 LONG   dh_subdrv()   ;/* SUBDRV Code Entry Point               */
 LONG   dh_uninit()   ;/* UNINIT Code Entry Point               */
 LONG   dh_select()   ;/* SELECT Code Entry Point               */
 LONG   dh_flush()    ;/* FLUSH Code Entry Point                */
 LONG   dh_read()     ;/* READ Code Entry Point                 */
 LONG   dh_write()    ;/* WRITE Code Entry Point                */
 LONG   dh_get()      ;/* GET Code Entry Point                  */
 LONG   dh_set()      ;/* SET Code Entry Point                  */
 LONG   dh_special()  ;/* SPECIAL Code Entry Point              */
 LONG   dh_ctl1       ;/* Reserved                              */
 LONG   dh_ctl2       ;/* Reserved                              */
 LONG   dh_ctl3       ;/* Reserved                              */
 LONG   dh_rlr        ;/* Pointer to Ready List Root            */
 LONG   dh_functab    ;/* Pointer to Driver Services Table      */
}
```

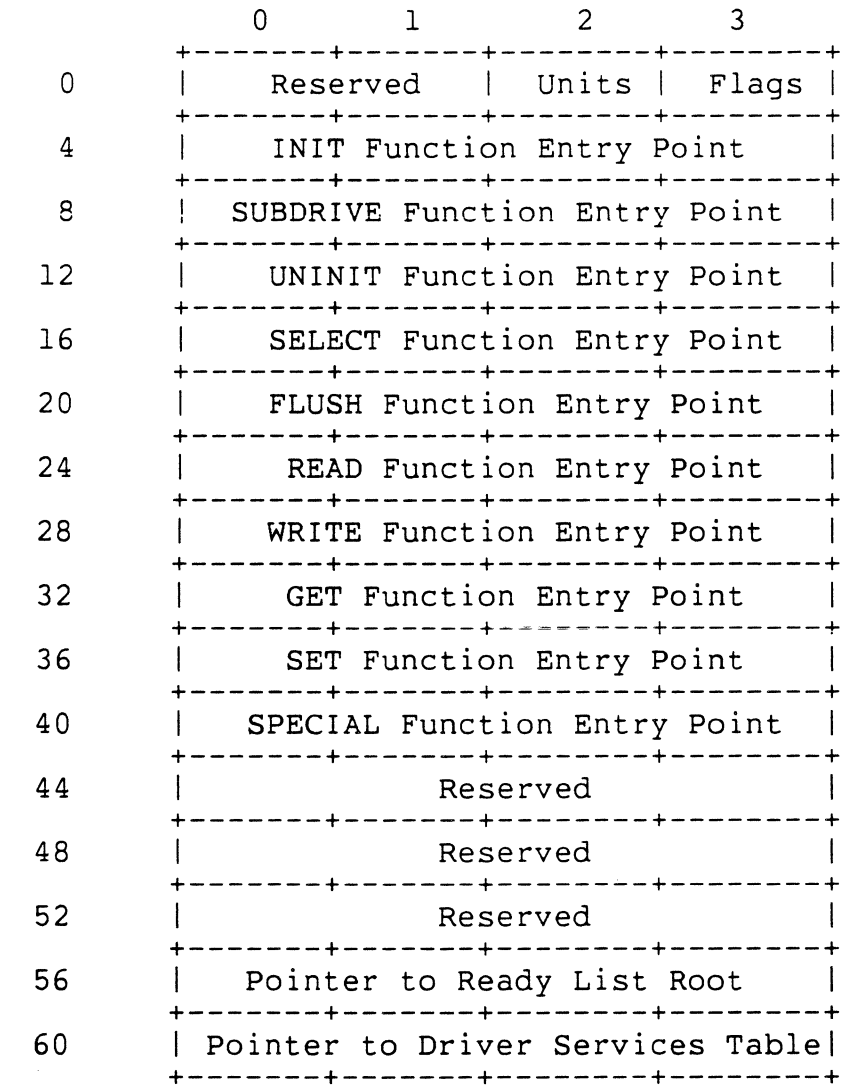


Figure 4-2. Driver Header Format

Table 4-1. Driver Header Data Fields

Data Field	Explanation
Units	This unsigned byte indicates the maximum number of units supported by this driver. A value of 0 indicates support for an unspecified number of units. INIT for unit 0 is called immediately following the first INSTALL. The unit number is incremented on each subsequent INSTALL.
Flags	The first four bits of this unsigned byte are used to specify driver interface information. See Table 4-2.
INIT	Address of the driver installation function called by the INSTALL SVC to initialize each unit of the driver.
SUBDRIVE	Address of the driver installation function that manages this driver's sub-driver information.
UNINIT	Address of the driver function called by INSTALL to uninitialize (remove) a driver unit.
SELECT	Address of the driver function that prepares a driver unit for subsequent I/O. SELECT is used in conjunction with the OPEN SVC.
FLUSH	Address of the driver function that "closes" a previously opened driver unit. The CLOSE SVC is mapped directly to this entry point.
READ	Address of the driver function called when a READ SVC has been specified.
WRITE	Address of the driver function called when a WRITE SVC has been specified.
GET	Address of the driver function called to fill a buffer with information about a driver unit.
SET	Address of the driver function called to control the SET function for the driver's units.

Table 4-1. (Continued)

Data Field	Explanation
SPECIAL	Address of the driver function called when a process requests special, device-specific functions.
Pointer to Ready List Root	Address in the FlexOS internal data area of the Process Descriptor for the process running before the current asynchronous I/O event was begun. This is the PDADDR of the process waiting for a system flag to be set which you pass to the FLAGSET driver service. This field is filled in by the Supervisor when the driver is installed.
Pointer to Driver Services Table	Address of a table containing the addresses of the FlexOS driver service routines (see Section 5). This table is used by the Driver Run-time Library linked with dynamic drivers. This field is also filled in by the Supervisor when the driver is installed.

The INIT, SUBDRIVE, and UNINIT driver installation functions are common to all driver types. See Section 4.4 for their descriptions. The remaining functions are driver-type dependent and described separately according to their resource manager in Sections 7, 8, 9, 10, and 11.

4.2.1 Driver Header Synchronization Flags

Three driver header flags indicate to the driver's resource manager if that driver can handle multiple I/O requests. The resource manager then controls the flow of requests to the driver depending upon the status of these bits. Another bit indicates whether the device controlled by the driver is 8- or 16-bit oriented. The following table lists the bit values:

Table 4-2. Driver Header Synchronization Flags

Flag	Value	Meaning
Bit 0:	0	I/O reentrant at the driver level
	1	Synchronize at the driver level
Bit 1:	0	I/O reentrant at the unit level
	1	Synchronize at the unit level
Bit 2:	0	I/O reentrant at the Resource Manager Level
	1	Synchronize at the Resource Manager Level
Bit 3:	0	Byte-oriented device
	1	Word-oriented device
Bit 4:	0	Use systems delimited read routine
	1	Use driver-supplied routine for delimited read requests

Flag bit 0 is the driver level synchronization flag. Set this flag to zero if the driver is able to handle multiple I/O requests simultaneously. If the driver must get I/O requests one at a time, set flag bit 0 to one.

Flag bit 1 is the unit level synchronization flag. As with the driver, set this flag to zero if the unit can handle multiple I/O requests simultaneously. Set this flag to one if the unit must complete one request before receiving another.

If the Resource Manager level interface flag, bit 2, is set, the resource manager allows the driver to perform a series of I/O operations for a single unit before permitting a different unit to perform another series of operations. Set this flag when a device being managed by this driver must be deactivated before another device can be used. If flag bit 2 is off, each unit can accept multiple outstanding I/O requests.

Flag bit 3 establishes a record size on a device as 1 or 2 bytes. This flag is used for delimited READs, to determine whether FlexOS will interpret a device's data as 8- or 16-bit characters.

Note: The GET function is not considered an I/O request and can be called at any time regardless of the synchronization flag value.

4.3 Entry Point Parameter Interface

The resource manager provides the driver installation and I/O functions with a 32-bit parameter and expects a 32-bit return code. The parameter is data or the address of a parameter block. The return code by definition indicates success with a positive value and failure with a negative value. The success return codes are described in the function descriptions below. See Appendix B in the FlexOS Programmer's Guide for the description of the FlexOS error codes. Error codes in the range of $-64x10^3$ to $-2x10^9$ are driver-type specific.

The C language entry point parameter interface convention is shown in Listing 4-2.

Listing 4-2. C Language Calling Convention

```
Calling Sequence:       ret = function(parm);

Function Interface:     LONG function(arg)
                        LONG arg;
                        {
                                LONG ret_code;
                                . . .
                                return(ret_code)
                        }
```

SELECT and SPECIAL return driver-type-specific error codes. INIT and FLUSH return driver-type-specific error codes through the driver's synchronous interface.

The READ, WRITE, and SPECIAL driver I/O functions are expected to return event masks through the driver's synchronous interface. These driver functions pass the completion code through the asynchronous interface.

4.4 Driver Installation Functions

This section describes the three driver installation functions common to all driver types: INIT, SUBDRIVE, and UNINIT. These functions map to the INSTALL SVC's options as follows:

- INIT is called to execute INSTALL options 1--load driver--and 2--add a unit.
- SUBDRIVE is called to execute INSTALL option 3--link two drivers.
- UNINIT is called to execute INSTALL option 0--remove a driver unit.

4.4.1 INIT--Initialize the specified driver unit

Parameter:

High Word	INSTALL Flags--see Table 4-4 below
Low Word	Unit number to be initialized

Return Code:

Success	**High word:** Return either 0 or sub-drive driver type value. A zero value indicates that initialization is complete--no sub-driver is required. At this point the unit is operational and mapped to a resource manager. If a sub-driver is required to make this driver operational, return the sub-driver's driver type value here. The driver type values are listed in Table 4-3 below. **Low word:** Return this driver's driver type value (see Table 4-3.
E_HARDWARE	Hardware not available
E_EXISTS	Specified unit already initialized
E_INIT	Driver unit could not be initialized
E_MEMORY	Could not allocate enough System Memory for the unit
E_xxx	Driver-specific type of error

The INIT driver function is called by the INSTALL SVC for each unit in the driver. The INIT driver function must initialize the specified unit's hardware. This function should also initialize any Mutual Exclusion Parameter Blocks (MXPBs) the unit will require, call FLAGGET for any flags to be used by the unit, establish the Interrupt Service Routine (ISR) vector through a call to the SETVEC driver service, and allocate System Memory for the unit. You can also use INIT to allocate buffers for initialized units.

The INSTALL flags selected by the user are specified in the high word of the entry parameter; the unit number is provided in the low word. The meaning for each flag value is listed in Table 4-4. INSTALL flag bit 8 is used by the Disk Resource Manager to determine if the disk device may have partitions. A disk device installed with partitions allowed cannot be formatted.

INIT must return the installed driver's type value. If the driver unit requires a sub-driver, INIT must also return the driver type of the required sub-driver. The following table lists the driver type values:

Table 4-3. Driver Type Values

Hex Value	Driver Type
0	Invalid or No Driver
1	Timer Driver
11	Pipe Driver
21	Disk Driver
31	Console Driver
38	Screen VDI driver
5x	Extension Drivers
61	Network Protocol Driver
62	Network Transport Driver
63	Network Transaction Server Driver
64	NET: Device Driver
65	Name Server Driver
71	Printer Driver
72	Serial Driver
78	Printer VDI driver
79	Metafile VDI driver
7D	Network Resource Manager
7E	DOS Clock Driver Emulator
7F	Null Device
81	Port Driver
82-FF	OEM Specific (Special)

Driver type values never end in zero; for example, 70 is an illegal driver type value. Zero in the second digit is reserved for resource managers.

Table 4-4. INSTALL Flags

Flag	Meaning
Bit 0:	0 = User Raw SET not allowed 1 = Raw SET allowed
Bit 1:	Reserved--must be 0
Bit 2:	0 = User Raw WRITE not allowed 1 = Raw WRITE allowed
Bit 3:	0 = User Raw READ not allowed 1 = Raw READ allowed
Bit 4:	0 = Exclusive access only 1 = Shared access allowed
Bit 5:	0 = Permanent device 1 = Removable device
Bit 6:	0 = DEVLOCKs not allowed 1 = DEVLOCKs allowed
Bit 7:	0 = Exclusive access allowed 1 = Shared access only
Bit 8:	0 = Device partitions not allowed 1 = Partitions allowed
Bits 9:	0 = Do not verify after disk writes 1 = Verify after disk writes
Bits 10-12	Reserved--must be 0

Table 4-4. (Continued)

Flag	Meaning
Bit 13:	0 = Do not force case to media default 1 = Force case to media default
Bit 14:	0 = Prefix substitution on load name 1 = Literal load name
Bit 15:	Reserved--must be 0

4.4.2 SUBDRIVE--Associate driver to a sub-driver

Parameter: Address of SUBDRIVE parameter block (see Figure 4-3 below)

Return Code:

Success **High word:** Set to either 0 or sub-drive. A zero value indicates that initialization is complete; the unit is operational and mapped to a driver. Return the required sub-driver's driver type value if another sub-drive is needed to complete the hardware interface. Table 4-3 lists the driver type values.

Low word: Set to 0.

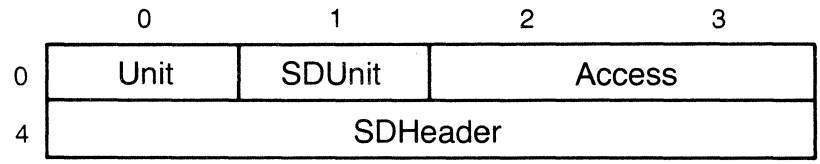


Figure 4-3. SUBDRIVE Parameter Block

Table 4-5. SUBDRIVE Parameter Block Data Fields

Field	Meaning
Unit	Driver unit that requires this sub-driver
SDunit	Sub-driver unit number
Access	The sub-driver's INSTALL access flags (see above). It is the higher-level driver's responsibility to honor these flags.
SDheader	Address of sub-driver's Driver Header

The SUBDRIVE function links one driver to another. Both drivers must be previously loaded and initialized. The user specifies in the INSTALL call which driver is to act as the resource manager and which driver is to act as the sub-driver. The SUBDRIVE parameter block provides you with the user's driver selections in the form of their unit numbers. Also provided in the parameter block are the subdriver's access flags (specified when that driver was installed) and the address of the sub-driver's driver header. The higher-level driver controls the sub-driver through the entry points at this address.

Sub-drivers are previously initialized units not currently in use. Once a unit has been declared a sub-driver, it cannot be addressed through its previous device name--it becomes dedicated to the higher-level driver. The higher-level driver becomes the sub-driver's resource manager.

4.4.3 UNINIT--Uninitialize the Specified Driver Unit

Parameter: Unit Number

Return Code: 0

The UNINIT driver function removes a driver unit from the system. UNINIT is responsible for releasing any system resources and for determining that the unit's hardware has been placed in a quiescent state before it is removed from the system.

Any files open to the unit are closed by the Resource Manager that controls the unit's driver. The Resource Manager also FLUSHs any buffers used by the unit.

UNINIT should not call a sub-driver's UNINIT function because FlexOS may map the sub-driver to another driver. It is important that UNINIT call the sub-driver's FLUSH function.

If every unit in a driver has been uninitialized, the driver can be removed from System Memory.

End of Section 4

Driver Services

FlexOS provides a number of services to drivers not available through the normal programmer interface. This section describes those services and where a driver would use them.

The driver service functions are described in C and are available to drivers whether they are dynamically installed or linked into the system image. Drivers linked with the system can access driver services directly. Drivers loaded from disk can link to a Driver Run-time Library, which indirectly calls the appropriate operating system routines.

The Driver Run-time Library accesses operating system functions by looking up the addresses of these routines in a table supplied by the operating system. The location of this table is placed into the driver header at the time the driver is installed.

The driver service functions are grouped into the following seven categories:

- the flag system (Section 5.1)
- Asynchronous Service Routines (5.2)
- device polling (5.3)
- memory management (5.4)
- critical regions (5.5)
- system process creation (5.6)
- interrupt service routines (5.7)

Driver services are listed alphabetically within each category.

Besides the driver services described below, drivers can make Supervisor calls through the Supervisor Interface (SUPIF). Section 6 describes how to call the Supervisor and the precautions you must observe.

5.1 Flag System

The FlexOS flag system acts as a logical interrupt system in which a process begins an asynchronous event and indicates that it will wait for the future completion of the event. The process that began the event then continues its execution asynchronously with respect to the hardware interrupt or process that completes the event. The flag system indicates the event's completion and awakens the original process.

The FlexOS flag system provides five driver service functions that enable the Supervisor to coordinate and acknowledge asynchronous events: FLAGGET, FLAGEVENT, FLAGSET, FLAGCLR, and FLAGREL. The Supervisor allocates flags to drivers with FLAGGET. FLAGEVENT enables a process to signal that it has begun an asynchronous event and will wait for the event's completion. The calling process is notified of the event's completion when another process or an Asynchronous Service Routine (ASR) calls FLAGSET. A process uses FLAGCLR to return a flag to its clear state and FLAGREL to release a flag back to the system.

A flag is similar to a binary semaphore - it is a single-event communication channel between two asynchronous routines. Unlike a binary semaphore, a process can use a flag to return a 32-bit value. The communication channel is established when a driver's INIT code calls FLAGGET. A flag can be in one of four states:

- unused - the flag has not been allocated
- clear - flag has been allocated but is not currently in use
- pending - A process has an event waiting for the I/O to complete
- completed - An I/O event has completed, but no process has performed a FLAGEVENT driver service function on it

There is a limit of 31 flags per process. The total number of flags in the system cannot be specified; FlexOS dynamically allocates new system flags as they are needed.

FLAGGET allocates a flag to the driver by searching for a system flag that is in the unused state and returning a flag number to be used by the driver in all future references to that particular flag. The Supervisor initializes the allocated flag by placing it in the clear state.

A driver must allocate a sufficient number of flags to handle the maximum number of asynchronous events that might occur at a given time. For example, a driver should allocate separate flags for READ operations and WRITE operations so that a READ and a WRITE event can be processed simultaneously.

For drivers using the flag system, an I/O operation takes place in the following sequence of events.

1. A process starts an I/O event by calling the appropriate driver function through the driver's synchronous interface. This begins an I/O operation that causes a hardware interrupt when the driver unit completes the I/O. At this point, the calling process actually runs the code in the driver.

2. The driver, under the process's control, then calls FLAGEVENT with a flag number to indicate which flag to mark as pending. The flag number was obtained through FLAGGET at the time the driver was initialized.

3. FLAGEVENT returns an event mask used by the calling process to wait for the completion of the event. The calling process passes the event mask to the WAIT SVC to wait for event's completion. If the event is already completed, i.e., FLAGSET has already been called and the flag is marked as completed, FLAGEVENT causes the flag to be marked as clear and the event itself is noted as completed.

4. When the I/O is completed, a hardware interrupt occurs that results in an Interrupt Service Routine (ISR) being executed. The ISR calls the DOASR driver service to schedule an ASR for execution. The ASR notifies the system that the event is completed by calling the FLAGSET driver service with the flag number, process descriptor address, and completion code as arguments.

If the original process has not caused FLAGEVENT to be called, FLAGSET sets the flag to completed, or, if the flag was pending, to clear. If the requesting process is waiting for the event, it is awakened. If the process canceled the event or the process was terminated, FLAGSET returns an error code.

Because of the asynchronous nature of FlexOS, it is possible for the I/O event to complete before the process starting the I/O has a chance to call the FLAGEVENT driver service. Once FLAGEVENT is called, the calling process returns from the driver code with the event mask as a return code.

When a flag is set to the clear state, the event it marked is placed on a list indicating it is waiting for the original process that called FLAGEVENT to perform the RETURN SVC. The flag can then be used by other processes, even though the event is not satisfied through RETURN.

The driver must remember the process descriptor address of the running process before the I/O event is actually started. This value is obtained through the Ready List Root (RLR) address field in the driver header. The driver must store this value locally until it is used by the ASR, or process, that calls FLAGSET.

The flag system driver services return error codes if a logic error has taken place. A FLAGEVENT performed on a flag in the pending state returns an E_UNDERRUN error; another process is already using this flag for another I/O event. A FLAGSET performed on a flag in the completed state returns an E_OVERRUN error; an I/O request completed and set a flag that was previously set. This error is also occurs when a process has not performed a FLAGEVENT function on the previous I/O event.

FlexOS returns an E_EMASK error if a process attempts, through a driver's FLAGGET call, to obtain more than 31 flags or when a driver calls FLAGEVENT when the calling process already has 31 outstanding I/O events.

When the driver is finished using a flag, it can release it with FLAGREL. FLAGREL places the flag back into the unused state. An error occurs if a flag is not in the clear state when the release is attempted (see FLAGCLR). If a driver is to use a flag frequently, the driver should not release the flag until its UNINIT code is executed.

When the communication between routines gets crossed up, the driver can force the flag into a clear state with the FLAGCLR driver service. This happens, for example, when the hardware produces spurious interrupts. FLAGCLR should be called before an I/O request is started.

Table 5-1 shows the results of the flag system driver service functions on system flags according to their state.

Table 5-1. Flag Operations and Flag States

Flag State	FLAGGET	FLAGREL	FLAGEVENT	FLAGSET	FLAGCLR
unused	clear	--	--	--	--
clear	--	unused	pending	completed	clear
pending	--	E_INUSE	UNDERRUN	clear	clear
completed	--	E_INUSE	clear	OVERRUN	clear

5.1.1 FLAGCLR--Clear a system flag

C Interface:

```
LONG    flagno;
LONG    retcode;

retcode = flagclr(flagno);
```

Parameters:

flagno System flag number to clear

Return Code: E_SUCCESS to indicate success

FLAGCLR forces a system flag into the clear state. In hardware environments where spurious interrupts might occur, the driver should call FLAGCLR before the process initiates the I/O operation.

5.1.2 FLAGEVENT--Return an event mask

C Interface:

```
LONG    swi;
LONG    flagno;
LONG    emask;

emask = flagevent(flagno,swi);
```

Parameters:

flagno	System flag number previously allocated by FLAGGET
swi	Software interrupt routine to be called when the event completes. This address is originally passed to the driver in the Supervisor call's parameter block. A zero value indicates that no swi was specified.

Return Code:

emask	Event mask. The calling process uses this value to wait for a subsequent FLAGSET on the given flag number.
E_UNDERRUN	Logic Error. A process is already waiting on this flag.
E_EMASK	No event mask is available. The calling process has 31 outstanding events. This error does not occur when FLAGEVENT is called by an ASR.

FLAGEVENT returns an event mask (emask) which allows the caller to wait for the setting of a system flag. It is assumed that the flag will be set asynchronously, however, in some instances the driver's synchronous code can call FLAGSET. The calling process can wait for the event through the WAIT SVC. The driver typically returns the event mask received through this driver service back to the resource manager that called the driver. The resource manager is responsible for either waiting for the event or returning to the calling process, depending on the type of call made.

5.1.3 FLAGGET--Allocate a system flag number

C Interface:

```
LONG    flagno;
flagno = flagget();
```

Parameters: None

Return Code:

flagno	Flag number
E_EMASK	31 flags have already been allocated to this process

The FLAGGET driver service allocates a system flag number. This operation is typically done in the driver's INIT code.

5.1.4 FLAGREL--Release a system flag

C Interface:

```
LONG    flagno;
LONG    retc;

retc = flagrel(flagno);
```

Parameters:

flagno	Flag number to be released

Return Code:

E_SUCCESS	Flag is released
E_INUSE	Flag is not in the clear state

FLAGREL releases a system flag number. This driver service is typically called from the driver's UNINIT code. FLAGREL returns an error if the flag to be released has not been previously cleared.

5.1.5 FLAGSET--Set a system flag

C Interface:

```
LONG    flagno;
LONG    pdaddr;
LONG    retcode;
LONG    retc;

retc = flagset(flagno,pdaddr,retcode);
```

Parameters:

flagno — System flag number as previously allocated by the FLAGGET driver service.

pdaddr — Process descriptor address of process waiting for this flag. Get this value from the RLR address in the driver header.

Note: This is NOT the pdaddr normally passed with parameter blocks into the driver entry points. The pdaddr normally indicated in the entry point Parameter Block is the process in whose memory the buffer belongs. The original calling process may be a different process.

retcode — Completion code for this operation.

Return Code:

E_SUCCESS	Flag is set
E_CANCELLED	Process canceled the FLAGEVENT
E_OVERRUN	Logic error - flag is already set

The FLAGSET driver service notes the completion of an asynchronous operation. The process that is waiting for this operation previously called--or is about to call--FLAGEVENT with the indicated flag number, from FLAGGET. If the process was aborted while waiting for this flag to be set or if the process canceled its WAIT, the E_CANCELLED error is returned.

5.2 Asynchronous Service Routines

Asynchronous Service Routines (ASRs) are routines within a driver's code that execute asynchronously to processes. FlexOS provides five driver service functions for executing ASRs:

- DOASR schedules an ASR for execution at the next dispatch.
- NEXTASR and EVASR schedule an ASR for execution upon the completion of an event.
- ASRWAIT suspends ASR execution until an event completes.
- DSPTCH forces a dispatch which results in the execution of all pending ASRs.

Because ASRs are run by the dispatcher, they have a higher priority than processes. At every process dispatch, the dispatcher checks to see if any ASRs have been scheduled to run. If there are one or more ASRs ready, it runs the first one to completion and checks for more. FlexOS schedules ASRs in priority order. ASRs of equal priority are scheduled on a first-come, first-serve basis. When there are no more ASRs, the dispatcher runs the next ready process.

The routines that respond to hardware interrupts are called Interrupt Services Routines, or ISRs. To allow FlexOS to respond to multiple interrupts, ISRs should be very short. Typically, an ISR schedules an ASR to complete the work required by the interrupting event. For example, in response to an interrupt, an ISR can call DOASR to schedule an ASR to perform I/O.

When an ASR starts an event through an SVC or a driver service, FlexOS returns an event number instead of an event mask. An event mask allows synchronous processes to wait for up to 31 different events; event numbers allow ASRs to wait for an unlimited number of events. Use multiple ASRs, each receiving its own event number, to wait on multiple events.

SVCs requiring an event mask parameter can be called from either an ASR or a process. If an ASR is calling, an SVC accepts event numbers instead of event masks.

When you receive an event number (or, in other contexts, an event

mask), you must call the RETURN SVC to clear the event from the system. This is true even for events used to synchronize ASRs and even if the STATUS SVC indicates the event is already complete. In response to an ASR, STATUS returns a 0 if the event is not complete. Any other value indicates completion.

To clear an event from the system, the ASR that generates an event number should call NEXTASR, EVASR, or ASRWAIT. For all three, the event is specified as a parameter in the call. For NEXTASR and EVASR, if you do not pass the event number in the call, you must store it in a global area accessible by the ASR scheduled. ASRs scheduled by NEXTASR or EVASR should call RETURN, passing the event number as the parameter. Call RETURN or STATUS through the Supervisor Interface (SUPIF) defined in Section 6. ASRs calling ASRWAIT should not call RETURN, the event is cleared and the completion code is returned by ASRWAIT.

ASRs scheduled by NEXTASR can wait on only one event before being scheduled. The event specification must be an event number; it cannot be a process's event mask. EVASR, on the other hand, accepts either a process event mask or an ASR event number. Like NEXTASR, EVASR is restricted to scheduling one ASR for an event completion. Use EVASR when you do not know whether the call is made from ASR context or process context.

ASRWAIT is a functional alternative to NEXTASR and EVASR that takes an event number (but not a event mask) and returns when the event completes. In general, it is more expensive in both CPU and memory utilization to use ASRWAIT versus next ASR or EVASR . However, if the current ASR stack is complex (that is, the current state of the logic is not easily reproduced), ASRWAIT can simplify reproducing it. ASRWAIT is a simple way to break up an ASR, since other ASRs can run before ASRWAIT returns.

An ASR may not call the WAIT SVC. Polling operations are strongly discouraged. ASRs may effectively perform a block by calling ASRWAIT or by chaining to another ASR with NEXTASR or EVASR, which executes after a specified event has occurred.

The priority of ASRs ranges from 0, the highest priority, to 255, the lowest. Digital Research recommends that most ASRs run at priority 200, allowing room for ASRs driven by real-time events to have higher priority than ASRs that need not be so timely.

ASRs run to completion. If a hardware interrupt occurs during the execution of an ASR, the dispatcher will continue execution of the ASR after the ISR completes. This occurs even if a higher-priority ASR is scheduled by the ISR.

ASRs can be disabled through the NODISP driver service, as described in Section 5.5. The DSPTCH driver service, described below, takes the currently running process out of context and forces all scheduled ASRs and poll routines to run.

5.2.1 ASRWAIT--Wait for event to complete

C Interface:

```
LONG    evnum;
LONG    ev_return;
BYTE    *stack_save_area;

ev_return = asrwait(evnum,stack_save_area);
```

Parameters:

evnum — Event number returned by SVC or driver service call

stack_save_area — Address of buffer for temporary stack storage

Return Code:

ev_return	Event's completion code
E_SUCCESS	No event number was specified

The ASRWAIT driver service suspends ASR execution until the specified event is complete. The event is designated by its number. The event must have been initiated by the ASR; it cannot be a process event mask. Specify a null event number to reschedule an eventless ASR. While the ASR is suspended, other ASRs are executed at the next dispatch.

The second ASRWAIT parameter is a buffer address. ASRWAIT copies a

portion of the dispatcher stack into this buffer and restores the stack from it. The area must be big enough to hold all of the stack used by this ASR since it was called, plus 50 to 100 bytes. No error or range checking is performed.

The ASR is rescheduled for the next dispatch after the event completes. The event's completion code is returned by ASRWAIT. Do not call the RETURN SVC after an ASRWAIT. When a null event number is specified, the ASR is rescheduled for the next dispatch and receives an E_SUCCESS event completion.

IMPORTANT: If you called MAPU before calling ASRWAIT, you must call MAPU again when the function returns to get the memory back.

5.2.2 DOASR--Schedule an ASR

C Interface:

```
VOID    asr_routine();
LONG    parm1;
LONG    parm2;
BYTE    prior;

doasr(asr_routine,parm1,parm2,prior);

VOID asr_routine(parm1,parm2)
LONG parm1;
LONG parm2;
{
/* perform activity */

return;
}
```

Parameters:

asr_routine	Address of ASR routine
parm1	First general parameter to pass to the ASR
parm2	Second general parameter to pass to the ASR
prior	ASR priority

Return Code:

E_SUCCESS	Successful operation
E_POOL	Out of memory

The DOASR driver service schedules an ASR for execution. Typically, Interrupt Service Routines call DOASR when the ISR needs more work done than can be performed in a timely manner from within the ISR. The ASR is placed in the ASR dispatch queue according to the priority parameter (0 = best, 255 = worst). All ASRs with equal priority are dispatched on a first-in, first-out basis.

ASRs run to completion before another ASR is run. While ASRs are running, hardware interrupts are enabled. This allows ISRs to run and to schedule other ASRs. When an ISR is complete, an interrupted ASR continues to run even if the ISR has scheduled a higher-priority ASR. After the ASR is complete, the scheduled ASR with the highest priority is run next.

5.2.3 DSPTCH--Force a dispatch

C Interface:

```
dsptch();
```

Parameters: None

Return Code: None

The DSPTCH driver service takes the currently running process out of context, reschedules it, runs all scheduled ASRs and poll routines, then brings the best priority, runnable process into context. The calling process is rescheduled as a running process. DSPTCH returns to the calling process when it comes back into context.

DSPTCH is useful in guaranteeing that all scheduled ASRs have run. The DOASR driver service, described above, does not force a dispatch but only schedules the ASR to run at the next dispatch.

5.2.4 EVASR--Schedule ASR from Process Context

C Interface:

```
VOID    asr_routine();
LONG    emask;
LONG    parm2;
BYTE    prior;

evasr(emask,asr_routine,parm2,prior);

VOID asr_routine(evnum,parm2)
LONG evnum;
LONG parm2;
{
   return;
}
```

Parameters:

asr_routine	Address of an ASR
emask	Process event mask or ASR event number
parm2	General parameter to ASR
prior	ASR priority

Return Code: None

The EVASR driver service schedules an ASR for dispatching upon the completion of the specified event. If EVASR is called in ASR context, this service is equivalent to NEXTASR with the event number as parm1. When EVASR is called in process context, EVASR converts the process event mask to an ASR event number, frees the process's event bit, and disassociates the event from the process. The new event number is passed as the first parameter to the ASR so that the ASR can call the RETURN SVC on the event number.

5.2.5 NEXTASR--Schedule ASR from an ASR

C Interface:

```
LONG    evnum;
VOID    asr_routine();
LONG    parm1;
LONG    parm2;
BYTE    prior;

nextasr(evnum,asr_routine,parm1,parm2,prior);

VOID asr_routine(parm1,parm2)
LONG parm1;
LONG parm2;
{
/* perform activity */

return;
}
```

Parameters:

evnum	Event number of event to wait for
asr_routine	Address of ASR routine
parm1	First general parameter to pass to the ASR
parm2	Second general parameter to pass to the ASR
prior	Priority of ASR

Return Code: None

The NEXTASR driver service schedules an ASR for dispatching upon completion of the event specified by the event number.

Call NEXTASR from within an ASR when an ASR needs to wait for the completion of an event. NEXTASR can be called only by the ASR that initiated the event upon which NEXTASR is waiting.

If you run an ASR that generates an event number, you must call NEXTASR to schedule an ASR to call the RETURN SVC, which clears the event from the system.

NEXTASR accepts only event numbers; an ASR cannot use an event mask generated by a process. The pending event must have been generated from within an ASR, not a process.

5.3 Device Polling

For devices not interrupt-driven, FlexOS supports the software mechanism of device polling. In single-tasking systems, these devices are usually polled with a hard CPU loop. However, this type of polling severely degrades the performance of a multitasking system. By using the POLLEVENT driver service, a device is polled periodically, allowing processes to run between polls.

Use POLLEVENT to emulate an asynchronous event when there is no hardware interrupt to determine completion of an event. POLLEVENT is not meant to replace the FLAGEVENT/FLAGSET method of communicating with an application, which is described in Section 5.1.

POLLEVENT is usually called from within an ASR. If called from an ASR, it returns an event number. The event number is used to perform a NEXTASR driver service, which performs a FLAGSET upon completion of the poll event.

Following the completion of the poll event, the driver must call the RETURN SVC to clear the event from the system.

The dispatcher calls the poll routine at process context switches. Therefore, poll routines run under the dispatcher process context. If the poll routine returns true (non-zero), the poll event is noted as completed. If a NEXTASR driver service was called based on the poll event, NEXTASR schedules an ASR to run.

5.3.1 POLLEVENT--Poll for event completion

C Interface:

```
WORD    poll_routine();
LONG    emask;
LONG    swi;

emask = pollevent(poll_routine,swi);
```

```
WORD    poll_routine()
{
/* check device */

if (device_ready)
return(-1);
else
return(0);
}
```

Parameters:

poll_routine	Address of poll routine. This routine returns 0 if the event is not complete. A non-zero return code indicates the poll event is complete. This routine is called at each process dispatch until the event is complete.
swi	Address of user software interrupt routine

Return Code:

emask	Event mask used to perform a WAIT based on this event (or a NEXTASR on this event number)

The POLLEVENT driver service establishes a poll routine which is called periodically to determine the completion of an event. POLLEVENT returns an event mask which allows the calling process to wait, through the WAIT SVC, for the software-determined event.

The poll routine is called under the dispatch-process context and is similar in nature to an ASR. If POLLEVENT is called by an ASR, an event number is returned instead of an event mask. Following completion of the polled event, the driver must call the RETURN SVC, through SUPIF, to clear the event from the system.

5.4 System Memory Management

FlexOS supports mapped and protected memory management units (MMUs). The following terms are used in the description of the FlexOS memory model.

- Physical Memory - all physically addressable memory in the system. This includes memory used for specific types of hardware, such as bit maps for video displays.
- Physical Space - the address space of Physical Memory. The addresses in Physical Space might be used when communicating with hardware, such as DMA controllers.
- User Memory - Physical Memory allocated for use by a particular process. Programs loaded from disk are placed into User Memory. Memory allocated through the MALLOC SVC is also placed in User Memory. Each process has its own User Memory.
- User Space - memory that can be addressed while running code in User Memory. Each process running code in User Memory is running in its own User Space. Each User Space is a separate address space. With supporting hardware memory protection, a process running in its User Space cannot address memory in another User Space. While running code in User Space, System Memory and Physical Memory are also not addressable.
- System Memory - Physical Memory allocated for use by the operating system. All drivers are loaded into System Memory. All memory allocated to drivers through the SALLOC function, described below, is also placed in system memory.
- System Space - System Space is the memory that can be addressed while running code in System Memory. At any point in time this space includes all System Memory as well as the User Memory of the currently running process.

  Some processes, called system processes, do not own User Memory. While system processes are running, there is no addressable User Memory.

All driver code resides in System Memory and therefore executes in System Space. This code is always running under a process context and therefore includes the process's User Memory. Asynchronous Service Routines (ASRs) run under the dispatch-process context and are considered system processes.

The addressing of User Memory is not necessarily the same in System Space as it is in User Space. For example, a buffer address supplied by an application while in User Space cannot be used directly while in System Space. The address must be translated into System Space before it addresses the same physical User Memory.

- User Address - an address that points to User Memory relative to User Space. While in System Space a User Address must be converted to a System Address before use.
- System Address - an address directly addressable while in System Space.

Figure 5-1 shows the relationship of User Space to System Space.

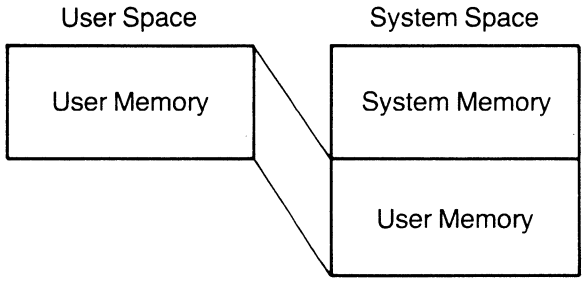


Figure 5-1. User Space and System Space

FlexOS supplies a number of driver service functions to drivers that facilitate the use of various types of addresses and also allocate and free system memory. These services are as follows:

- SADDR - converts a User Address in User Memory to a System Address.
- UADDR - converts a System Address in User Memory to a User Address. An error is returned if the System Address points to System Memory.
- PADDR - converts a System Address to a Physical Address.
- MAPU - allows a process to change the User Memory currently mapped in system space to another process's User Memory. The calling process loses access to its own User Memory until the UNMAPU function is called.
- UNMAPU - restores a process's User Memory.
- MLOCK - locks the current User Memory in Physical Memory. This prevents moving the memory to another physical location.
- MUNLOCK - allows User Memory to be swapped out to disk or moved to another physical memory location. System Addresses of User Memory may change while the memory is unlocked. System Addresses of User Memory should be converted to a User Address before the MUNLOCK function is called and converted back to a System Address after the MLOCK function has been called.
- MRANGE - checks the start and length of a buffer in User Memory to verify it is within the process's current User Space.
- SALLOC - allocates System Memory from the free pool of Physical Memory. This is the same pool used by applications.
- SFREE - frees System Memory and places it back into the free pool of Physical Memory.
- MAPPHYS - maps physical memory not in the free pool into System Space and returns a System Address for that memory. This is used to address "device memory" such as bit maps or read-only Memory. If the device memory is already mapped to System Space, the System Address is returned. MAPPHYS should be called only at INIT time.

Resource Managers call the driver entry points to I/O functions with pointers to buffers that can be either User or System Addresses. If the buffer resides in User Memory the pointer is a User Address. If the buffer resides in System Memory, the pointer is a System Address. The resource manager sets a flag to indicate whether the address is in User or System memory. Along with the flag, drivers receive the process descriptor address (PDADDR) of the process that owns the buffer's memory.

A User Address is not directly usable until it is converted to a System Address. The User Address is relative to a particular process. SADDR converts a User Address into a System Address for the currently addressable User Memory.

When drivers pass a User Address to ASRs or other processes, it must be passed as a User Address and process descriptor address pair. This allows the ASR or process to call the MAPU driver service to assume the original User Memory, then call the SADDR driver service to obtain the System Address of the correct Physical Memory. Passing a System Address of User Space to an ASR or another process results in addressing the wrong Physical Memory or a memory violation upon use of that address.

Driver entry points are called with User Memory locked in Physical Memory. A driver has the option of unlocking the memory to allow moving the memory to another physical location.

Moving memory might be done by the memory manager during garbage collection. If the driver calls MUNLOCK to unlock User Memory, all System Addresses that refer to User Memory become invalid. Before MUNLOCK is called, UADDR must be used to convert to user addresses all System Addresses that refer to User Memory. These converted addresses cannot be used either by the driver or hardware until the User Memory is locked into Physical Memory through the MLOCK driver service. The driver can then use SADDR to convert User Addresses to System Addresses and Physical Addresses for use by the driver and its hardware.

For all SVCs for which the user program specifies a buffer, FlexOS does buffer range checking to ensure that all buffers sent to drivers are contiguous in physical memory and legal. An exception to this is the SPECIAL SVC, where a buffer is not assumed but might be sent to

the driver by an application. In this case, the driver must perform its own range checking though the MRANGE driver service.

A driver can call the SALLOC driver service to allocate System Memory to be used by the driver for buffers and other memory resources. This is usually done in the driver's INIT code. Allocating memory at INIT time allows a driver's load image to be small. It also allows a driver to handle an arbitrary number of units by allocating memory as INIT is called for each unit.

Memory allocated through SALLOC should be freed with the SFREE driver service in the driver's UNINIT code.

SALLOC takes memory out of the Transient Program Area (TPA), which is the same Physical Memory pool from which User Memory is allocated during program loading. Thus, memory allocated by SALLOC is not available to loadable programs.

Before using SALLOC, you should also consider that more memory than requested might be taken out of the Transient Program Area (TPA). This potential for wasted memory ranges from 512 to 16K bytes. The exact amount depends on the granularity of the MMU's mapping ability or, in segmented architectures, on the minimum size of a segment. The amount of wasted memory is also related to the minimum fragmentation allowed by implementation-dependent memory management routines.

5.4.1 MAPU--Map another process's User Memory

C Interface:

```
LONG    pdaddr;

ret=mapu(pdaddr);
```

Parameters:

pdaddr — Process descriptor address of process whose User Memory is to be mapped. No checking is done to verify that the pdaddr is valid.

The driver usually receives this address through the PDADDR field of a parameter block passed through one of the driver's entry points. The buffer is specified in the same parameter block.

Return Codes:

E_SUCCESS	Successful operation
emask	Designated process is currently swapped out. It will be swapped in asynchronously to the calling process. A WAIT on this event mask returns when the specified process's memory is in place.

MAPU removes the calling process's current User Memory and replaces it with the indicated process's User Memory.

5.4.2 MAPPHYS--Map Physical Memory

C Interface:

```
BYTE    *saddr
MAPPB   *parmblk;
WORD    type;

saddr = mapphys(parmblk,type);
```

Parameters:

parmblk	Address of map parameter block describing physical memory (see Figure 5-2)
type	0 = code 1 = data

Return Code:

saddr	System Address of mapped physical memory

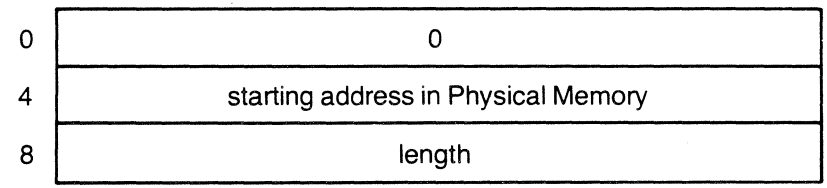


Figure 5-2. Map Parameter Block

MAPPHYS puts the specified Physical Memory into System Memory. The specified Physical Memory cannot be part of the TPA. MAPPHYS should be called only once, at the time a driver is initialized. If the memory is to contain executable code, the type parameter must be zero. If the type parameter is zero, mapped memory cannot be modified as data.

Use MAPPHYS to obtain System Addresses of device memory, such as bit maps, or other memory not intended for the direct use of applications. An example of this type of memory is read-only memory that can be accessed only from System Space.

5.4.3 MLOCK-- Lock the User Memory

C Interface: mlock();

Parameters: None

Return Code: None

MLOCK locks the current User Memory in Physical Memory. MLOCK prevents moving the memory to another physical location. The Supervisor automatically locks memory whenever an application calls an SVC.

The driver is responsible for matching MLOCK and MUNLOCK calls. FlexOS maintains a count of the number of locks in force and will not unlock the memory until the number of MUNLOCK calls matches the number of MLOCK calls applied to the User Memory.

5.4.4 MRANGE--Perform range checking

C Interface:

```
BYTE    *start;
LONG    length;

retc = mrange(start,length);
```

Parameters:

start	Starting User Address of buffer in User Memory
length	Number of bytes in the buffer

Return Code:

E_SUCCESS	Legal buffer
RD_ONLY	Buffer is read only
SYS_SPC	Buffer is in system space
	These two bits indicate the designated buffer status on a successful return from MRANGE ().
E_RANGE	Range error

The MRANGE driver service allows a driver to verify that a buffer in User or System Memory does not violate memory protection before the buffer is used. Following a successful return from MRANGE, a driver can call a DMA device knowing a memory violation trap will not occur.

5.4.5 MUNLOCK--Unlock User Memory

C Interface: munlock();

Parameters: None

Return Code: None

MUNLOCK unlocks the current User Memory in Physical Memory. MUNLOCK allows moving the memory to another physical location. The Supervisor automatically locks the current User Memory whenever an application calls an SVC.

The driver is responsible for matching MLOCK and MUNLOCK calls. FlexOS maintains a count of the number of locks in force and will not unlock the memory until the number of MUNLOCK calls matches the number of MLOCK calls applied to the User Memory. If memory is unlocked by a driver, the driver must lock the memory before returning to the calling process.

5.4.6 PADDR--Convert address: System to Physical

C Interface:

```
BYTE    *physadr;
BYTE    *sysadr;

physadr = paddr(sysadr);
```

Parameters:

sysadr	System Address to convert

Return Code:

physadr	The physical address of the specified System Address

PADDR converts a System Address to a physical address. Use PADDR to convert a buffer address in System Space to a physical address and then give the address to a hardware device, such as a DMA controller.

5.4.7 SADDR--Convert address: User to System

C Interface:

```
BYTE    *usradr;
BYTE    *sysadr;

sysadr = saddr(usradr);
```

Parameters:

usradr	Address of User Memory from User Space

Return Code:

sysadr	System Address of converted User Address

SADDR converts a User Address into a System Address relative to the current User Memory. The User Address of another process's User Memory can be converted to a System Address by first calling the MAPU driver service, described above, and then SADDR.

5.4.8 SALLOC--Allocate System Memory

C Interface:

```
LONG    length;
BYTE    *sysadr;

sysadr = salloc(length);
```

Parameters:

length	Number of bytes to allocate

Return Code:

sysadr	Address of memory block allocated in System Memory
0	No memory available to satisfy the request

The SALLOC driver service allocates System Memory from the TPA.

5.4.9 SFREE--Free System Memory

C Interface:

```
BYTE    *sysadr;

ret = sfree(sysadr);
```

Parameters:

sysadr	Address of previously allocated System Memory

Return Code:

E_SUCCESS	Successful operation
E_MEMORY	Illegal memory reference

SFREE frees memory allocated through the SALLOC driver service. The address to be freed must be one returned through SALLOC.

5.4.10 UADDR--Convert address: System to User

C Interface:

```
BYTE    *sysadr;
BYTE    *usradr;

usradr = uaddr(sysadr);
```

Parameters:

sysadr	Previously converted System Address of User Memory

Return Code:

usradr	User Space address of User Memory
E_MEMORY	sysadr not in User Memory

UADDR converts a System Address to a User Address. The System Address must point into User Memory. An error occurs if the System Address points into System Memory.

5.4.11 UNMAPU--Restore User Memory

C Interface: unmapu();

Parameters: None

Return Code: None

UNMAPU restores the calling process's User Space. MAPU allows a process to map temporarily another process's User Memory into System Space. UNMAPU removes the current User Memory and restores the process's own User Memory into System Space. If the calling process is a system process, no User Memory is mapped.

5.5 Critical Regions

FlexOS supplies routines to allow a driver to set up critical regions without turning off hardware interrupts. FlexOS recognizes three types of critical regions:

- Mutual exclusion regions - allow a driver to restrict multiple processes from accessing a resource or data structure. Driver services are MXINIT, MXEVENT, ASRMX, MXREL, and MXUNINIT.
- No-abort regions - guarantee that a particular process will not abort while in the no-abort region. Drivers can use this type of region to ensure that a set of tasks will be completed by the calling process. The driver services are NOABORT and OKABORT.
- No-dispatch regions - guarantee that no other processes or ASRs will run while the system is in the no-dispatch region. Typically, this region is used where a resource, such as a linked list, is accessed by many processes from many different locations in the code. A no-dispatch region guarantees that no other process will access the resource while it is being used by the current process. You can also use this type of region where the calling process cannot "hang," waiting for a mutual-exclusion region. The no-dispatch region should be used with care, because it directly affects the response time of a process to an external event. The driver services are NODISP and OKDISP.

FlexOS allows drivers to set up mutual exclusion regions to protect data structures from multiple processes accessing them, without turning off hardware interrupts. These mutual exclusion regions can also be used to protect non-reentrant code and make it a serially reusable resource.

The mutual exclusion primitives are similar to a semaphore system, where a process must get a semaphore before using a resource. The semaphore is released when the resource is no longer needed. If the semaphore is in use by another process, the calling process receives an event mask which can be used to wait for the semaphore. When multiple processes wait for the same semaphore, the requests are queued on a a first-come, first-serve basis.

FlexOS maintains the semaphore, its current owner, and a list of processes waiting for the semaphore in a data structure called the Mutual Exclusion Parameter Block, or MXPB. Drivers interface with MXPBs through routines described below.

A driver creates an MXPB through the MXINIT driver service. MXINIT returns a 32-bit value that identifies the MXPB for future use. Typically, a driver stores this value in its data area and accesses the value whenever a process attempts to use a protected resource. The driver usually calls MXINIT from its INIT code.

FlexOS provides two driver services for obtaining an MXPB: MXEVENT and ASRMX. You use MXEVENT when you are in process context; use ASRMX when in ASR context. For both functions, the caller becomes the owner if the MXPB is not in use. If the MXPB is owned by another process when you call the service, MXEVENT returns an event mask. ASRMX returns an event number when the MXPB is owned by any process, including the calling process.

Use WAIT or EVASR with the event mask to wait for the MXPB to be released. The WAIT SVC is only valid when you are in process context. If you are in ASR context, use NEXTASR or ASRWAIT to reschedule the ASR upon the release of the MXPB. When you use WAIT, EVASR, or NEXTASR, you must call the RETURN SVC to clear the event after you receive control of the MXPB. Do not call RETURN, however, if you use ASRWAIT; the event number is cleared and the completion code returned by the function.

Use the MXREL driver service to release the MXPB when you are done with it. If you acquired the MXPB with ASRMX, you must make the MXREL call from within ASR context. If you call MXEVENT from within a process's context, you must call MXREL from within the same process's context. This is almost impossible to do if you go into ASR context between the MXEVENT and MXREL calls. Consequently, most calls to obtain an MXPB should be made from within ASR context.

If a process is aborted while it owns an MXPB, the MXPB is automatically released.

A driver can remove an MXPB from the system through the MXUNINIT driver service. An error is returned if the MXPB is in use. MXUNINIT is usually called in the driver's UNINIT code.

5.5.1 ASRMX--Obtain MXPB ownership

C Interface:

```
LONG    mxid;
LONG    retc;

retc = asrmx(mxid);
```

Parameters:

mxid	MXPB ID as returned by MXINIT

Return Codes:

E_SUCCESS	MXPB obtained
evnum	Event number. MXPB is owned.

The ASRMX driver service obtains ownership of an MXPB. If the MXPB is already owned, either by the calling process or another process, an event number is returned. Use this number in a NEXTASR or ASRWAIT call to schedule ASR execution to wait upon the release of the MXPB.

5.5.2 MXEVENT--Obtain MXPB ownership

C Interface:

```
LONG    mxid;
LONG    retc;

retc = mxevent(mxid);
```

Parameters:

mxid	MXPB ID as returned by MXINIT.

Return Code:

E_SUCCESS	MXPB obtained
emask	Event Mask--MXPB owned by another process

The MXEVENT driver service obtains ownership of an MXPB. If the MXPB is already owned, that is, if the object to be locked is in use, the return value will be an event mask that can be used to wait, through the WAIT SVC, for ownership.

5.5.3 MXINIT--Create an MXPB

C Interface:

```
LONG    mxid;

mxid = mxinit();
```

Parameters: None

Return Code:

mxid	New MXPB's ID

The MXINIT driver service returns a 32-bit value identifying a Mutual Exclusion Parameter Block (MXPB) to be used with the MXEVENT and MXREL driver services. MXINIT is usually called from the driver's INIT code.

The MXPB is an abstract structure to the driver writer, who passes the structure pointer to the MXEVENT and MXREL driver services. FlexOS allocates space for the MXPB out of System Memory.

5.5.4 MXREL--Release an MXPB

C Interface:

```
LONG    mxid;

retc = mxrel(mxid);
```

Parameters:

mxid	MXPB ID as returned from MXINIT

Return Code:

E_SUCCESS	Successful operation
E_OWNER	Calling process is not owner of MXPB

The MXREL driver service releases an MXPB and therefore exits a mutual exclusion region. If another process is waiting for the MXPB, it receives ownership of it.

5.5.5 MXUNINIT--Remove an MXPB from the system

C Interface:

```
LONG    mxid;
LONG    retc;

retc = mxuninit(mxid);
```

Parameters:

mxid	MXPB ID as returned by MXINIT

Return Code:

E_SUCCESS	Successful operation
E_INUSE	MXPB currently in use.

The MXUNINIT driver service removes an MXPB from the system. In response to MXUNINIT, FlexOS deletes the specified MXPB from the MXPB list and frees the memory containing the MXPB for other uses. MXUNINIT is usually called from a driver's UNINIT code.

5.5.6 NOABORT--Enter no-abort region

C Interface: noabort();

Parameters: None

Return Code: None

The NOABORT driver service begins a no-abort region, that is, NOABORT disables abort routines. A no-abort region prevents abort routines from executing as long as the region is active. As soon as abort routines are enabled (see the OKABORT driver service) all pending abort requests for the process are attempted. In the case of multiple NOABORT calls, each NOABORT call must be matched by an OKABORT call to reenable abort routines.

5.5.7 NODISP--Enter a no-dispatch region

C Interface: nodisp();

Parameters: None

Return Code: None

The NODISP driver service begins a no-dispatch region and thereby disables dispatches of processes and ASRs. Execution of NODISP allows you to disable dispatching of user tasks and ASRs until OKDISP is executed. In the case of multiple NODISP calls, each NODISP call must be matched by an OKDISP call to reenable process and ASR dispatches.

5.5.8 OKABORT--Exit no-abort region

C Interface: okabort();

Parameters: None

Return Code: None

The OKABORT driver service ends a no-abort region and thereby enables abort routines. Any abort routines called during the no-abort region are executed.

5.5.9 OKDISP--Exit a no-dispatch region

C Interface: okdisp();

Parameters: None

Return Code: None

The OKDISP driver service ends a no-dispatch region and therefore enables dispatching of processes and ASRs.

5.6 System Process Creation

You create system processes with the PCREATE function driver service. A system process runs in System Space and owns no User Memory. In your PCREATE call you specify the address in System Memory where the process is to start execution, the stacksize, the priority, and the name of the process. PCREATE lets you send two parameters to the process as it starts execution.

PCREATE allocates a process data space, including a system stack and initializes the process name, priority, and other data. PCREATE then initializes the stack to contain the specified parameters and finally schedules the new process to run. FlexOS starts the process at the address you pass as a parameter to PCREATE.

The process has the full context and flexibility of any other process in the system, with the sole exception that the process does not own any User Memory.

5.6.1 PCREATE--Create a system process

C Interface:

```
LONG    pid;
VOID    start();
BYTE    *name;
BYTE    prior;
LONG    stacksize;
LONG    parm1;
LONG    parm2;
LONG    emask;

emask = pcreate(&pid,start,name,prior,stacksize,parm1,parm2);

VOID start(parm1,parm2)
LONG parm1;
LONG parm2;
{
/* first line of "C" Code that the new process  */
/*      will execute follows:                   */

            . . .

/* Terminate the system process                 */

s_exit(0);
}
```

Parameters:

&pid	Address of a 32-bit variable that will be modified by this routine to contain the process ID of the new process. This value is needed to abort the process and to obtain information about it.
start	Address of first instruction the new process will execute
name	Address of process name. The name is null-terminated. If the string is longer than eight bytes, only the first eight bytes are used.
prior	Initial process priority, ranging from 0 to 255. The guidelines for selecting the priority are:

1	INIT process
2-31	High-priority system process requiring immediate response to external events
32-63	System process
64-128	Undefined
129-199	High-priority user process
200	Normal user process
201-254	Low-priority user process
255	Idle process

stacksize	Size of the system stack for the new process. If this value is 0, a default value is used that allows SVCs to be called.
parm1	First 32-bit parameter to new process
parm2	Second 32-bit parameter to new process

Return Code:

emask	Event mask that can be used to wait, through the WAIT SVC, for the termination of the created process. Upon completion of the WAIT, that is, when the process has terminated, use the RETURN SVC to obtain the process's exit code.
E_MEMORY	Cannot allocate System Space for this process.

5.7 Interrupt Service Routines

Interrupt Service Routines (ISRs) are established through the SETVEC function described below. FlexOS transfers control to the Entry Point of the ISR as though it were calling a C routine.

It expects back one of two possible WORD values:

- true (1), meaning dispatching is required
- false (0), meaning no dispatching is required

Ideally, the ISR should work along the following lines:

1. The driver's INIT function sets up the ISR vector through a SETVEC call. SETVEC is described below. The driver's SELECT function enables hardware and software interrupts.
2. The driver's READ or WRITE code starts an operation that results in an interrupt, which transfers control to the ISR. If the device is of a type that does not require immediate service, the ISR might do no more than execute the DOASR function. If the device requires immediate service, e.g., a serial driver or a disk driver, the ISR might set up a DMA transfer or input or output the next data byte, then execute a DOASR to clean up.
3. The ISR determines whether or not a significant event, that is, an event which requires dispatching, has occurred. If it has, the ISR should return a true value; otherwise, the ISR should return a false value.

The following guidelines should be kept in mind when using ISRs.

- FlexOS takes care of stack switching, dispatch scheduling, and CPU-dependent interrupt resets.
- Interrupts are disabled upon entry to the ISR. Interrupts can be subsequently enabled by the ISR to allow nested interrupts.
- FlexOS saves all registers for the ISR. Therefore, it is not necessary for the ISR itself to preserve any registers.

- To allow FlexOS to respond to other interrupts in a timely way, ISRs should be kept as short as possible. In most cases, the majority of the work should be carried out by an ASR.
- Forcing a dispatch by returning "true" has overhead. If the external event is not required to be handled in real time, a "false" should be returned, even if the DOASR driver service function has been called. If the dispatch is not forced, the ASR will run at the next dispatch. The worst that could happen is that the ASR would have to wait for the next tick.

5.7.1 SETVEC--Set interrupt vector to ISR

C Interface:

```
LONG    intno;
WORD    isr_routine();
WORD    prev_isr();

prev_isr = setvec(isr_routine,intno);

WORD isr_routine()
{
/* service interrupt condition */

/* schedule ASR */
do_asr( ... );

/* dispatch or not depending on how critical */
/* it is to run the ASR if one was scheduled */

if (dispatch)
return(-1);   /* TRUE = force dispatch */
else
return(0);    /* FALSE = no dispatch */
}
```

Parameters:

isr_routine	Address of Interrupt Service Routine
intno	Interrupt vector number

Return Code:

E_SUCCESS	Successful operation
prev_isr	Address of previous ISR routine. A return code of zero indicates that this is the first time SETVEC has been called for this interrupt vector. If prev_isr is non-zero, SETVEC has already been called for this vector.

The SETVEC driver service sets the specified interrupt vector to execute the specified Interrupt Service Routine. The physical interrupt vector will actually refer to an operating system routine which sets up the ISR environment, that is, saves registers.

Once the ISR returns, the registers are restored and the operating system routine either restores the environment and returns to the interrupted process or forces a dispatch to occur. If a dispatch is forced, the interrupted process is rescheduled and will run at a later time, according to its priority.

End of Section 5

Supervisor Interface

This section describes how drivers interface to the FlexOS Supervisor.

6.1 Supervisor Entry Point

Drivers can use the Supervisor Calls (SVCs) available to user programs and described in the FlexOS Programmer's Guide. Drivers linked with the system directly access the operating system services. Drivers loaded from disk link to a Driver Run-time Library which indirectly calls the appropriate operating system services.

Do not call an SVC which forces the driver to be reentered. This can result in a deadlock situation.

ASRs cannot call SVCs that result in a process waiting. If this occurs, the Dispatcher cannot schedule any tasks, including ASRs. This results in a system crash.

Calls passed to the Disk, Console, and Network Resource Managers might cause a process to wait. The Pipe Resource Manager is designed to be used by ASRs. However, even when performing operations on pipes, you must call SVCs asynchronously so that event masks are returned, instead of performing a wait.

ISRs cannot call SVCs.

When drivers access SVCs through SUPIF:

- The Supervisor does not perform buffer range checking.
- Parameter blocks are always 32 bytes and must be in System Memory.

- Bit 1 of the mode field in the parameter block must be set to 1 if the addresses in the parameter block are User Addresses. The mode field is the first byte (lowest address) of the parameter block. Set bit 1 to 0 if the parameter block addreses are System Addresses. Note that bit 0, the least significant bit, is the asynchronous bit.

The specific interface to SUPIF is described below.

6.1.1 SUPIF--Make a Supervisor call

C Interface:

```
WORD    funcno
LONG    param;
LONG    ret;

ret = supif(funcno,param);
```

Parameters:

funcno	SVC number
param	32-bit parameter, typically a parameter block address

Return Code:

ret	32-bit return code

The SUPIF driver service allows code within System Space to make SVC calls. The specific SVC numbers, parameters, and expected return codes are specified in the FlexOS Programmer's Guide.

End of Section 6

Console Drivers

This section describes the specific driver interface to character console drivers. It provides an overview of console drivers, discusses the FRAME and RECT data structures, and defines entry and return parameters for each console driver I/O function.

7.1 Console Driver Overview

A console driver is composed of one or more driver units. The Console Resource Manager (RM) manages each unit as a separate physical console device. Each physical console device has two components: a video display and a keyboard. There is no explicit limit to the number of physical consoles or the number of console drivers managed by the Console RM. Limits depend only on memory constraints.

FlexOS supports a standard console environment model independent of physical console device type. As a console driver writer, you must translate this model to your specific physical device. All device-dependent code is in the console driver.

The foremost consideration in writing a console driver is performance: the console must appear lively to the user. To help you implement your drivers, two sample drivers, a serial driver for a Zenith® Z-29 VDT and a character/bit-mapped driver for an IBM PC/AT, are distributed with FlexOS.

The sample drivers take advantage of the sub-driver architecture, described in Section 2.6. The use of this architecture eases the task of implementing console drivers for similar types of console devices.

Each console driver manages a class of console devices. The console driver can directly control each physical console it manages or can control individual physical consoles through sub-drivers. Each unit of a console driver corresponds to a single physical console device. FlexOS does not require that a driver's physical consoles be of a similar type. However, to conserve memory for both driver code and

data, it is desirable to have all consoles of similar types controlled by the same driver.

For example, consider a FlexOS system with three terminals; two serial consoles and one memory-mapped character console. This system should have two console drivers; one driving the memory-mapped console, while the other drives the two serial consoles. The serial consoles can be of different types if sub-drivers are used to hide differences between them. Figure 7-1 illustrates this example system.

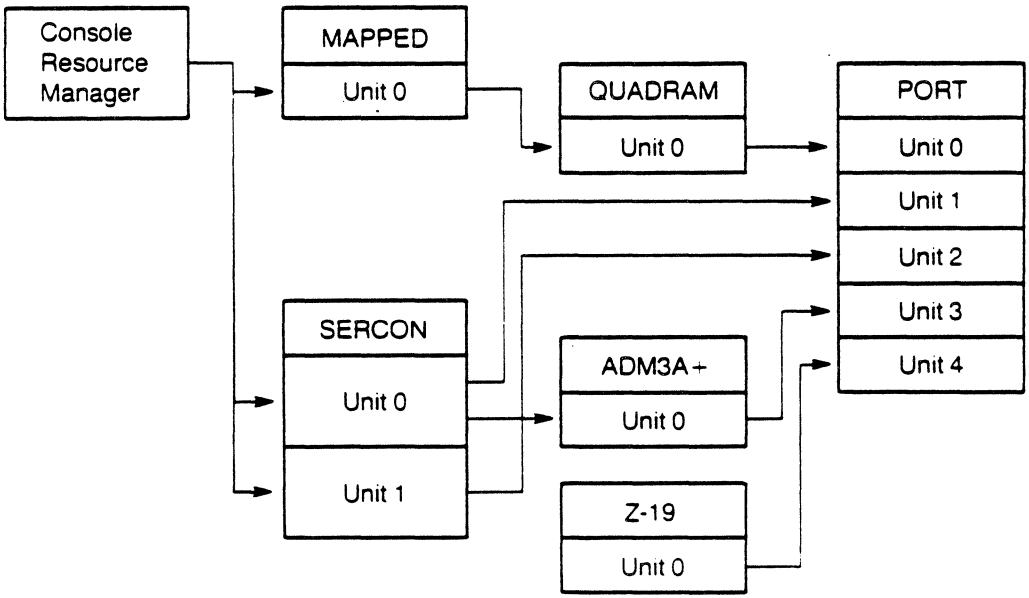


Figure 7-1. Console Drivers

In Figure 7-1, the console driver MAPPED supports a single physical console. MAPPED interfaces directly to the keyboard and screen hardware. It uses a subdriver to interface to the character-mapped hardware associated with this physical console. The driver QUADRAM handles the video display and keyboard interfaces, but is written to be

independent of the actual port hardware. QUADRAM uses the port driver PORT as a sub-driver to interface with the port hardware.

The driver SERCON handles much of the higher-level interface to serial consoles independent of the type of terminal. SERCON calls the sub-drivers ADM3A+ and Z-19 for the screen and keyboard interfaces. ADM3A+ and Z-19 handle the specific terminal interfaces and use PORT as a sub-driver to interface with specific port hardware.

7.2 The FRAME and RECT Structures

A FRAME is a logical representation of a screen. It is a three-dimensional structure consisting of one or more planes of character cells, with one byte per character cell. Each plane consists of either a two-dimensional byte array or a single byte which the Console RM uses to define all the bytes in the plane.

A FRAME's height and width are defined in terms of character columns and rows of its planes. A FRAME's depth is defined in terms of the number of planes in the FRAME.

On a FRAME, a rectangle can be described that descends through all of the FRAME's planes. This piece of the FRAME is a data structure called RECT. COPY and ALTER (see Section 7.4) manipulate FRAMEs by acting on RECTs. Figure 7-2, below, illustrates a FRAME with a RECT descending through its planes.

7.2.1 Planes

FlexOS defines parameters for three planes: the character (plane 0), the attribute (plane 1), and extension plane (plane 2). Figure 7-2, above, depicts these planes. Support for planes 1 and 2 is optional. These planes are defined as follows:

- Character plane - consists of alphanumeric characters. This plane uses an 8-bit character set defined on a per-country basis. This plane supports two-byte characters, such as KANJI, through the implementation of the extension plane (see below).

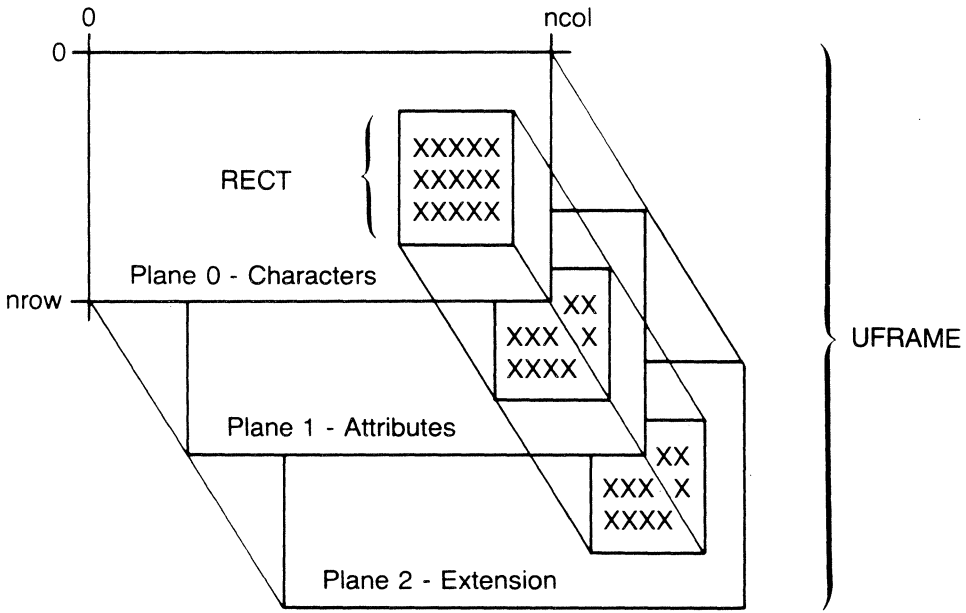


Figure 7-2. FRAME and RECT

- Attribute plane - describes the display characteristics of the characters in the character plane. Each byte in the attribute plane defines the foreground color, background color, color intensity, and blink status (on or off) for the corresponding character cell in plane 0.

The attribute byte is formatted as follows for monochrome and color video display drivers:

Bits 0-2	Foreground color
Bit 3	Intensity
Bits 4-6	Background color
Bit 7	Blink

For video displays supporting the blink attribute, set the blink bit in a given attribute byte to cause the corresponding character to blink.

The three-bit foreground and background color fields are defined as follows:

low bit	blue
middle bit	green
high bit	red

Use of the three color bits provide the following eight colors:

Table 7-1. Colors Defined in Attribute Byte

3-bit Value	Color
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	Light Gray

The foreground color, specified by bits 0-2, is modified by the intensity bit, bit 3. When the intensity bit is set, the low nibble of the attribute byte allows for the following colors:

Table 7-2. Foreground Colors with Intensity Bit Set

Low Nibble Value	Color
8	Dark Gray
9	Light Blue
AH	Light Green
BH	Light Cyan
CH	Light Red
DH	Light Magenta
EH	Yellow
FH	White

The attribute byte has the same format for monochrome video displays as for color. Certain color selections effect monochrome display output. For example, when the foreground color is black and the background color white, a monochrome display will appear in reverse video.

- Extension plane - allows support for alternate character set or other extensions to the standard FRAME. Implement this plane if you intend to support foreign languages. Extension plane bytes have the following format:

Bit 0	Cell type
Bit 1	Cell number
Bits 2 & 3	Reserved
Bits 4-7	OEM extension

Cell type (bit 0) determines whether the character corresponding to an extension byte is one byte or two. A one-byte character, such as an ASCII character, takes up one character position on the screen. A two-byte character, such as a KANJI character, takes up two character positions. Bit 0 is set to zero to display one-byte characters; Bit 0 is set to 1 to display two-byte characters.

Cell number (bit 1) indicates whether a corresponding byte in the character plane is either: a) the first part of a two-byte character or a one-byte character or b) the second part of a two-byte character. Bit 1 is set to 0 when the corresponding character plane byte is the first part of a character or when displaying one-byte characters. Bit 1 is set to 1 when a corresponding character plane byte is the second part of a character.

You can customize the OEM extension field (bits 4-7) for your own purposes. This field allows the implementation of alternate character sets. This field is set to zero when the FlexOS standard character set is supported.

7.2.2 FRAME Types

There are three types of FRAMEs: a user FRAME (UFRAME), a virtual FRAME (VFRAME), and a physical FRAME (PFRAME). The driver writer creates the PFRAME and VFRAME; FlexOS defines the UFRAME.

The UFRAME is a device-independent representation of a console screen used by applications. It is based on the model of the IBM PC video map. The FlexOS Programmer's Guide describes the UFRAME for the applications programmer.

The VFRAME is the storage form of a virtual console, as defined by the console driver writer. The Console RM stores a virtual console's current screen image in the VFRAME that the driver creates. The VFRAME can be created to be different sizes, where size is measured in rows and columns. FlexOS makes no assumptions regarding the VFRAME's format.

The Console RM calls the SPECIAL entry point to create and delete VFRAMEs as virtual consoles are created and deleted. See "SPECIAL Entry Point" for a description of the FlexOS support for creating and deleting VFRAMEs.

The Console RM uses the WRITE and COPY/ALTER functions to update the VFRAME. The WRITE entry point updates a VFRAME and returns a "dirty region" that allows the Console RM to determine the portions of the VFRAME that must be copied to the PFRAME. The VFRAME's design should provide for fast VFRAME to PFRAME COPY.

Through the SPECIAL functions 0, 2, and 3, FlexOS supports systems

whose VFRAME is modeled after the video map of an IBM PC. These SPECIAL functions are described below, under "SPECIAL Entry Point."

The PFRAME is a direct representation of the physical screen. Like the VFRAME, the PFRAME is defined by the console driver writer. Any change to the PFRAME must be reflected on the physical screen. The console driver must be able to COPY/ALTER the physical screen and therefore needs a copy of the PFRAME in memory. Most memory-mapped screen devices already have a PFRAME: the mapped memory itself.

If your PFRAME does not follow the IBM PC video map model, you must translate between your PFRAME and an IBM PC-type of PFRAME.

Most implementations of FlexOS console drivers simplify the FRAME transformation by defining the VFRAME and PFRAME to have the same memory representation. For serial devices, all three types of FRAME can have the same representation.

7.3 Console Driver Entry Points

Like all FlexOS drivers, console drivers consist of a driver header and entry points for driver functions. Section 4 describes the general format of a FlexOS driver. The entry points to a console driver are SELECT, FLUSH, COPY/ALTER, WRITE, SPECIAL, GET, and SET.

The SELECT function activates the keyboard. SELECT contains a pointer to the keyboard Asynchronous Service Routine (ASR) which calls the Console RM asynchronously. This ASR buffers the characters to be used by an application. FlexOS calls the SELECT entry point for keyboard information; there is no READ entry point.

FLUSH deactivates keyboard activity by disabling interrupts. Typically, the driver activates and deactivates keyboard hardware with SELECT and FLUSH calls.

WRITE and COPY/ALTER act on FRAMEs. These entry points are called to perform updates to the physical console. COPY/ALTER replaces READ in the standard FlexOS Driver Header. The Console RM performs range checking of the UFRAME before it calls COPY/ALTER.

Note: The WRITE and COPY/ALTER entry points are called from ASRs

and therefore can never wait. These functions return a zero if the operation is completed successfully. WRITE returns an event mask if the driver must wait for an event, or an error code if an error occurs.

The Console RM calls the SPECIAL entry point to

- create and delete virtual consoles (VFRAMEs)
- convert VFRAMEs to conform to an IBM® PC video map model
- convert VFRAMEs from IBM PC model back to original form
- change VFRAME configuration

The SPECIAL functions need not be implemented if you do not support virtual consoles or if you do not support PC DOS applications.

GET provides information on the physical console. SET changes the country code for a console for systems that support foreign character sets. The SET function is optional.

7.4 Console Driver I/O Functions

7.4.1 SELECT--Activate keyboard

Parameter: Address of SELECT parameter block

Return Code: E_SUCCESS Operation was successful

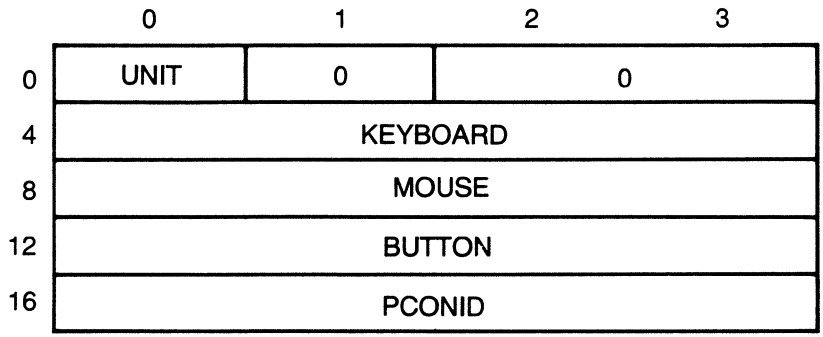


Figure 7-3. SELECT Parameter Block

Table 7-3. Fields in SELECT Parameter Block

Field	Description
UNIT	Driver unit number
KEYBOARD	Address of the keyboard ASR. Use this address in your DOASR or NEXTASR call to transfer a character from the keyboard to the input buffer. You must translate the character into the FlexOS 16-bit input character set (see Appendix A). If the keyboard generates toggle characters, they should always be passed to the keyboard ASR. When you call DOASR or NEXTASR, the first parameter is PCONID and the second is the character received.
MOUSE	Address of the mouse ASR. Use this address in your DOASR or NEXTASR call to transfer the change in the mouse position. When you call DOASR or NEXTASR, the first parameter is PCONID and the second is the address of the mouse movement packet with the delta x and delta y values.
BUTTON	Address of the button ASR. Use this address in your DOASR or NEXTASR call to indicate the mouse button pressed. When you call DOASR or NEXTASR, the first parameter is PCONID and the second is a long indicating the button pressed where bit 31 represents the leftmost mouse button, bit 30 represents the next button to the right, and so forth.
PCONID	Physical console identifier for the driver unit being SELECTed. The Console RM gives the PCONID to the driver of this unit. The driver must pass the PCONID to the keyboard ASR to allow it to identify which physical console is sending information.

The Console RM calls the SELECT function to initialize the keyboard. Once SELECT has been called, the keyboard is considered live.

Typically, SELECT turns on the hardware interrupts. FlexOS calls only the SELECT entry point for input data.

The interrupt vector itself is usually initialized in the INIT entry point at the time the driver is installed. Initialize the interrupt vector with the SETVEC driver service. Section 5.7 explains SETVEC and offers guidelines for using ISRs under FlexOS.

In response to an interrupt, the keyboard interrupt service routine should use the DOASR driver service to schedule the keyboard ASR. You must use the ASR provided in the SELECT parameter block, so be sure to save this address in your routine.

Non-interrupt-driven console devices use the POLLEVENT driver service to establish a poll routine to receive a physical input. POLLEVENT is described in Section 5.3.

The console driver must perform any necessary translation of hardware data to logical information that can be used by FlexOS. If translation is required, the driver, upon receiving a physical input, calls the DOASR driver service from an ISR to schedule an ASR to perform physical-to-logical translation. The translation should not be done in the ISR itself. Such translation might include translation to the FlexOS 16-bit character set. The FlexOS standard input character set is defined in Appendix A.

When translation is finished, the driver should call DOASR again to schedule the keyboard ASR, passing the translated information. The Console RM places the data into the present keyboard owner's input buffer.

Call the DOASR driver service with a priority of 200 when scheduling a translation ASR and the keyboard ASR.

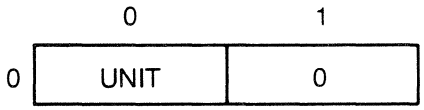
7.4.2 FLUSH--Deactivate keyboard

Parameter: Address of FLUSH parameter block

Return Code:

E_SUCCESS	Operation was successful
IOERROR	Console driver error code

The Console RM calls FLUSH with the address of the following parameter block:



The UNIT field contains the driver unit to be FLUSHed.

The FLUSH function is the reverse of SELECT. It must perform all operations (either hardware or software) required to stop the physical input of characters and/or interrupt sources. If a console driver owns a sub-driver, it must call the sub-driver's FLUSH function to make the sub-driver quiescent.

7.4.3 COPY/ALTER--Modify a RECT

Parameter: Address of COPY/ALTER parameter block

Return Code:

E_SUCCESS Operation was successful

COPY and ALTER share the same entry point. You determine which operation to perform from the OPTION field in the parameter block. Figure 7-4 illustrates the format of the COPY/ALTER parameter block. The fields are described in Table 7-4. The FRAME and RECT data structures referenced in the parameter block are illustrated following the parameter block description.

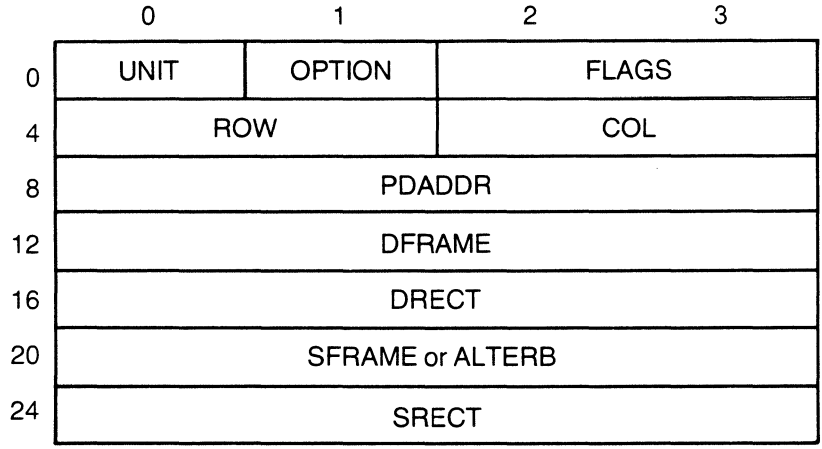
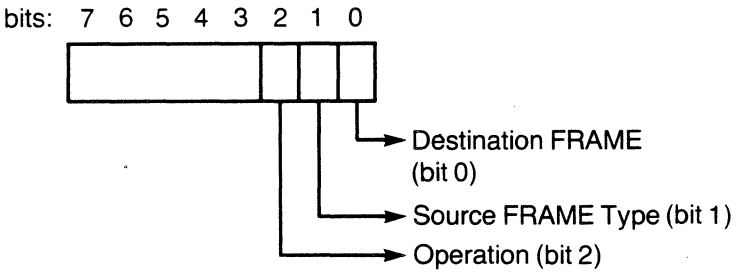


Figure 7-4. COPY/ALTER Parameter Block

Table 7-4. Fields in COPY/ALTER Parameter Block

Field	Description
UNIT	Driver unit number
OPTION	Bit map of operation and FRAME types:



Where FRAME type is: 0 - VFRAME/PFRAME
1 - UFRAME

Operation is: 0 - COPY source to destination
1 - ALTER destination

RECT and FRAME addresses for a UFRAME are always in User Memory. You must convert them to system addresses (see the SADDR driver service). The driver creates the PFRAME and VFRAME; consequently, these FRAMEs are in System Memory.

FLAGS Bit map of flag usage.

Bit 0: 1 = Modify plane 0
0 = Do not modify plane 0

Bit 1: 1 = Modify plane 1
0 = Do not modify plane 1

Table 7-4. (Continued)

Field	Description
	Bit 2: 1 = Modify plane 2 0 = Do not modify plane 2
	Bits 3-6: Reserved
	Bit 7: 1 = This is top virtual console. Update the cursor position when this bit is set. 0 = This is not the top virtual console.
	Bits 8-11: Reserved
	Bit 12: 1 = This call is to move the cursor only. Only UNIT, FLAGS, ROW, and COL have meaning. 0 = All fields have meaning
	Bit 13: 1 = This is an update of a dirty RECT passed from WRITE driver function. 0 = This is not a RECT passed from WRITE.
	Bit 14: 1 = Update PFRAME and VFRAME. This means the virtual console is the same size as the physical, windowed full-screen, and on top. 0 = Update as indicated with option. If VFRAME is modified, the Console RM updates windowed RECTs on PFRAME as appropriate.
	Bit 15: 1 = Buffer is in User Memory. Use the SADDR driver service to convert User to System Address before accessing this address. 0 = Buffer in System Memory.

Table 7-4. (Continued)

Field	Description
ROW	With COL, defines current cursor position. When COPY/ALTER is exited, this is where cursor should be.
COL	With ROW, defines current cursor position. When COPY/ALTER is exited, this is where cursor should be.
PDADDR	Process descriptor address of user process whose memory contains the UFRAME. This may not be the calling process for FLAGEVENT and FLAGSET. Get the pdaddr of the process accessing the COPY/ALTER entry point from the RLR Address field in the Driver Header.
DFRAME	Destination FRAME. Address of UFRAME, virtual console identifier (VCID) of VFRAME, or 0 if PFRAME.
DRECT	Address of destination RECT describing region in DFRAME.
SFRAME or ALTERB	For COPY operation, address of source RECT describing region in SFRAME. For ALTER operation, address of ALTERB; a six-byte array indicating the alteration of the destination RECT. The array is arranged as follows: alterb[0] = character plane AND alterb[1] = character plane XOR alterb[2] = attribute plane AND alterb[3] = attribute plane XOR alterb[4] = extension plane AND alterb[5] = extension plane XOR
SRECT	Source FRAME. Address of UFRAME, virtual console identifier (VCID) of VFRAME, or 0 if PFRAME. Not used for ALTER operation. The VCID is returned from SPECIAL function 0, Create VFRAME. SPECIAL function 0 is described later in this section.

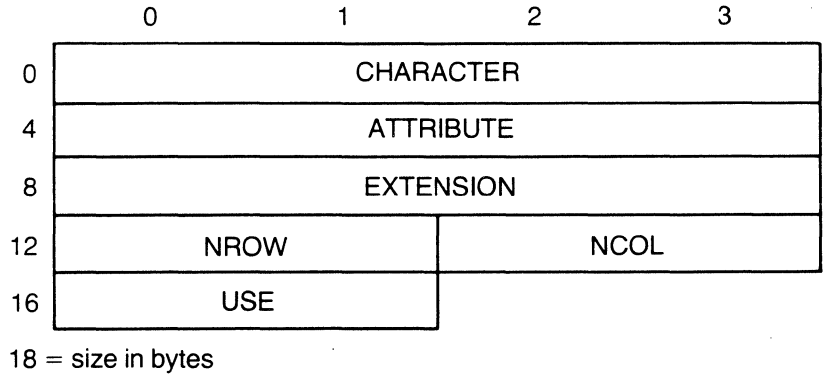


Figure 7-5. FRAME Structure

Table 7-5. FRAME Fields

Field	Description
CHARACTER	Address of the character plane of this FRAME
ATTRIBUTE	Address of the attribute plane of this FRAME
EXTENSION	Address of the extension plane of this FRAME
NROW	Number of rows; indicates each FRAME's height.
NCOL	Number of columns; indicates each FRAME's width.

Table 7-5. (Continued)

Field	Description
USE	Bit map describing how the three plane fields are used. When the bit value is 0, the byte at the address specifies the value for each element in the plane. Bit 0:1 = CHARACTER addresses a two-dimensional array of bytes making up the character plane. 0 = CHARACTER addresses a single byte. Bit 1:1 = ATTRIBUTE addresses a two-dimensional array of bytes making up the attribute plane. 0 = ATTRIBUTE addresses a single byte. Bit 2:1 = EXTENSION addresses a two-dimensional array of bytes making up the extension plane. 0 = EXTENSION addresses a single byte.

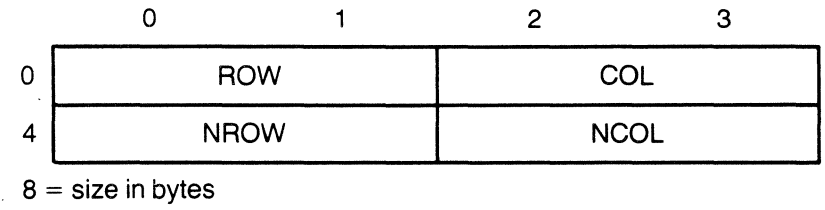


Figure 7-6. RECT Structure

Table 7-6. RECT Fields

Field	Description
ROW	Row position of the upper left corner of the RECT. The upper left corner, as specified by ROW and COL, is the reference point for a RECT.
COL	Column position of the upper left corner of the RECT.
NROW	Number of rows, indicating the RECT's height.
NCOL	Number of columns, indicating the RECT's width.

COPY copies the bytes from the region described by the source RECT into the region described by the destination RECT. If the RECTs are different sizes, the driver should trim them to the same size, using the upper left corner of each RECT as a reference point. The driver should store trimmed RECTs locally. If both RECTs are on the same FRAME and they overlap, care should be taken to copy the RECT in the appropriate direction.

ALTER alters the destination RECT by performing a logical AND operation with a specified AND byte and a logical XOR operation with a specified XOR byte on each byte of a given plane. Separate AND and XOR bytes are specified for each plane in the ALTERB array, defined above. The FLAGS parameter determines which planes will be effected.

The ALTER driver function allows an application to set, clear, complement, or leave unchanged any bit in the raw bytes of the destination RECT. This function should enable an application to perform such operations as clearing a portion of the screen, displaying strings of identical characters in different parts of the screen, or changing the attributes of a portion of the display without effecting the character or extension plane.

7.4.4 WRITE--Write data to VFRAME

Parameter: Address of WRITE parameter block

Return Code:

E_SUCCESS	Operation was successful
Event Mask	Event mask if operation will complete asynchronously

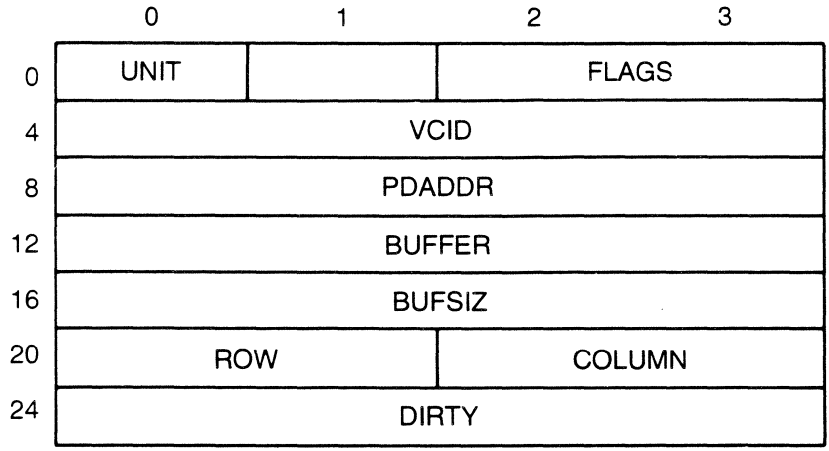


Figure 7-7. WRITE Parameter Block

Table 7-7. Fields in WRITE Parameter Block

Field	Description
UNIT	Driver unit number
FLAGS	Bit map of flags
	Bit 7: 1 = This is the top virtual console. 0 = This is not the top virtual console. This bit determines whether the WRITE operation should affect the cursor position on the physical console. If bit 7 is set, update the cursor position in the WRITE operation. If it is not set, do not update the cursor.
	Bit 14: 1 = Update PFRAME. 0 = Update VFRAME or PFRAME as indicated. Set Bit 14 to 1 when the virtual console is full screen, on top. This setting allows optimized use of screen-editing commands in 16-bit character set.
	Bit 15: 1 = Buffer is in User Memory. 0 = Buffer is in System Memory. If bit 15 is set to 1, use the SADDR service to convert User to System Address before accessing this address.

Table 7-7. (Continued)

Field	Description
VCID	Virtual console identifier of VFRAME or, if writing to the PFRAME, 0. The VCID is returned from SPECIAL function 0, described later in this section.
PDADDR	Address of process in whose memory BUFFER (see below) resides. This is not necessarily the calling process as needed in the FLAGEVENT and FLAGSET driver services. Obtain the pointer to the calling process's pdaddr from the RLR field in the Driver Header.
BUFFER	Address of buffer of 16-bit characters used to update the indicated VFRAME or PFRAME.
BUFSIZ	Size in bytes of BUFFER. This is not the number of characters. To obtain the number of characters divide BUFSIZE by two.
ROW	Current cursor row position on which to start placing characters.
COLUMN	Current cursor column position on which to start placing characters.
DIRTY	Address of structure to be filled in by WRITE indicating new cursor position and dirty region. Figure 7-8 illustrates the format of the dirty region.

	0	1	2	3
0	Cursor ROW		Cursor COL	
4	Dirty ROW		Dirty COL	
8	NROWS		NCOLS	

Figure 7-8. Dirty Region Format

Cursor ROW and Cursor COL indicate the cursor's new location. Dirty ROW and Dirty COL are the coordinates of the upper left corner of the dirtied RECT. NROWS and NCOLS indicate the size of the dirtied RECT.

WRITE updates a VFRAME with a specified buffer of 16-bit characters. The driver should write the string buffer at the specified cursor position. Before returning you must update the cursor position (the ROW and COLUMN values) and fill in the dirty region data structure.

7.4.5 SPECIAL Entry Point

The Console RM calls the SPECIAL entry point to perform the following functions:

- Special Function 0: Create a virtual console (VFRAME)
- Special Function 1: Delete a virtual console (VFRAME)
- Special Function 2: Convert a VFRAME to a PCFRAME
- Special Function 3: Convert a PCFRAME to its original form (inverse of Function 2)
- Special Function 4: Change VFRAME configuration

The SPECIAL functions operate on VFRAMEs, the storage form of a virtual console (see Section 7.2.2). A driver that does not support virtual consoles should return a not implemented (E_IMPLEMENT) error to the Console RM when the SPECIAL entry point is called.

Note: A PCFRAME is a VFRAME with the following characteristics:

- **For a character-mapped screen:** A 25 row by 80 column display with the characters arranged in character/attribute pairs
- **For a bit-mapped screen:** A 200 by 640 pixel display

FlexOS defines the UFRAME to allow a fast transformation to an IBM PC-type of PFRAME.

SPECIAL Functions 2 converts a VFRAME from its original form to a model that replicates an IBM PC video map. SPECIAL Function 3 converts the VFRAME from the IBM PC model back to its original form. These functions are provided to allow applications that poke the IBM PC model video map to run in a multiple-virtual console environment.

SPECIAL Function 0--Create a VFRAME

Parameter: Address of SPECIAL parameter block

Return Code:

VCID — An identifier of this VFRAME for use in the WRITE, COPY/ALTER and SPECIAL functions 1-3. This is typically the address of an internal data structure known by this driver. The VCID cannot be 0.

0 — Error. The Console RM assumes a memory allocation error has occured. A negative error code cannot be returned here since addresses may look like a negative number.

E_IMPLEMENT — Virtual consoles are not implemented

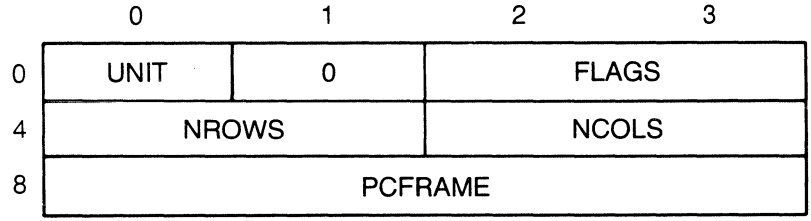


Figure 7-9. SPECIAL Function 0 Parameter Block

Table 7-8. Fields in SPECIAL Function 0 Parameter Block

Field	Description
UNIT	Driver unit number
0	One byte set to zero
FLAGS	Bit map of flag usage Bit 0: 1 = bit-mapped device 0 = character-mapped device Bits 1-6: Reserved Bit 7: 1 = creating a PFRAME 0 = creating a VFRAME
NROWS	Number of rows in VFRAME
NCOLS	Number of columns in VFRAME
PCFRAME	Preset to zero. If this field is non-zero, the value is the address of an IBM PC-compatible character or bit map.

The Console RM calls SPECIAL Function 0 to create a VFRAME. When the PCFRAME field contains the address of an IBM PC-type video map, use the FLAGS field to determine whether the display is bit- or character-mapped.

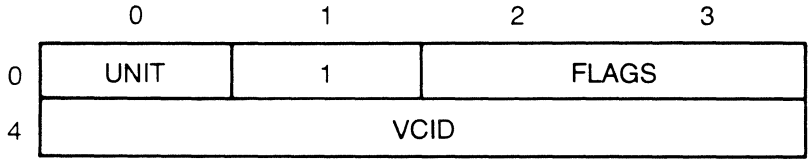
SPECIAL Function 1--Delete a VFRAME

Parameter: Address of SPECIAL parameter block

Return Code:

E_SUCCESS Operation was successful
E_IMPLEMENT Virtual consoles are not implemented

The Console RM calls SPECIAL function 1 to delete a VFRAME and provides the following information in the parameter block:



The UNIT field contains the driver unit number. 1 is the SPECIAL function number. The word at offset 2 is set to zero. VCID is the VFRAME identifier of the VFRAME to delete. The VCID is returned from SPECIAL function 0.

SPECIAL Function 2--Initialize a PCFRAME

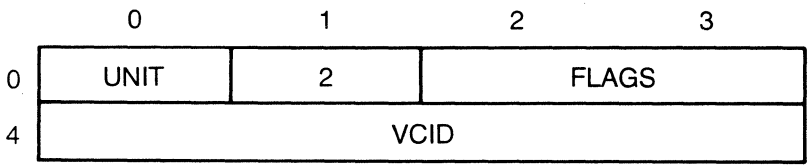
Parameter: Address of SPECIAL parameter block

Return Code:

address	Video map address
0	Operation not allowed

The Console RM calls SPECIAL function 2 when an application attempts to write directly to an IBM PC video map. Use this function to convert the specified VFRAME to a facsimilie of an IBM PC video map and return the address of the replacement video map. FlexOS directs subsequent console output to the video map address returned.

The SPECIAL parameter block provided by the Console RM is formatted as follows:



The UNIT field contains the driver unit number. 2 is the SPECIAL function number. The Console RM uses flag bit 0 only. If it is set to 1, it indicates that the application attempted to output to a bit-mapped display (address B800:0 in the IBM PC). If bit 0 is set to 0, the application attempted to output to a character-mapped display (address B000:0 in the IBM PC). The VCID identifies the VFRAME to convert.

SPECIAL Function 3--Revert to VFRAME

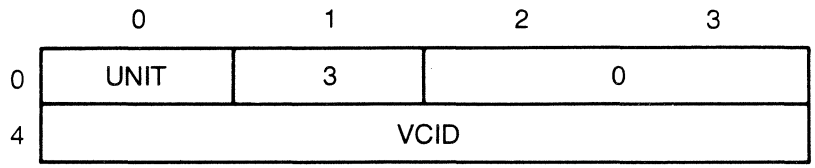
Parameter: Address of SPECIAL parameter block

Return Code:

address	Original VFRAME video map address
0	Operation not allowed

The Console RM calls SPECIAL function 3 when an application has stopped writing directly to an IBM PC video map. Use this function to convert the designated PCFRAME back to the VFRAME originally created.

The SPECIAL Function 3 parameter block is formatted as follows:



The UNIT field contains the driver unit number. 3 is the SPECIAL function number. The word at offset 2 is set to zero. VCID identifies the PCFRAME to convert back to a VFRAME.

SPECIAL Function 4--Change VFRAME Configuration

Parameter: Address of SPECIAL parameter block

Return Code:

address	Video map address
E_IMPLEMENT	Virtual console change does not match physical capabilities of console device
E_MEMORY	Not enough memory to change virtual console configuration

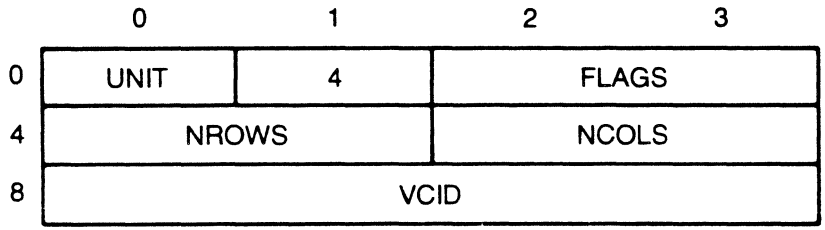


Figure 7-10. SPECIAL Function 4 Parameter Block

The Console RM calls SPECIAL Function 4 to reconfigure a virtual frame to a new configuration; for example, to convert a 80 x 25 black and white character screen to a 320 x 200 pixel color graphics screen.

Table 7-9. Fields in SPECIAL Function 4 Parameter Block

Field	Description
UNIT	Driver unit number
4	SPECIAL function number
FLAGS	Bit map of flag usage Bit 0: 1 = Graphics virtual console requested 0 = Character virtual console requested Bits 1-2: Reserved Bit 3: 1 = Color display 0 = Black and white display Bits 4-15: Reserved
NROWS	VFRAME's number of rows (bit 0 = 0) or height in pixels (bit 0 = 1)
NCOLS	VFRAME's number of columns (bit 0 = 0) or width in pixels (bit 0 = 1)
VCID	Identification number of VFRAME to reconfigure.

7.4.6 GET--Provide physical console description

Parameter: Address of GET parameter block

Return Code:

E_SUCCESS	Operation was successful
IOERROR	Driver-specific error code

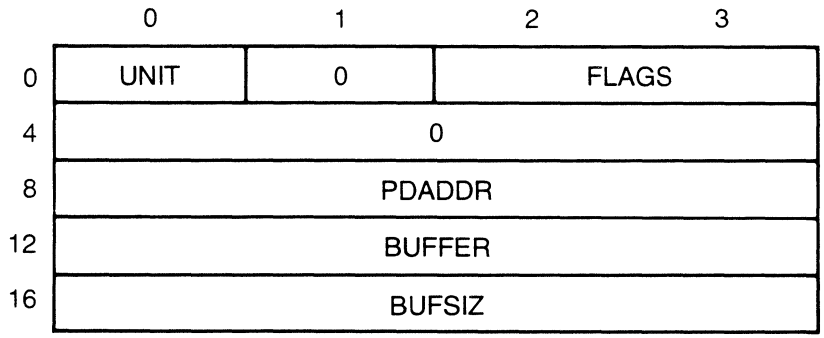


Figure 7-11. GET Parameter Block

Table 7-10. Fields in GET Parameter Block

Field	Description
UNIT	Driver unit number
0	Byte set to zero
FLAGS	Bit map of flag usage Bit 15: 1 = Buffer in User Memory 0 = Buffer in System Memory
0	Long word set to zero
PDADDR	Address of process descriptor of process owning BUFFER
BUFFER	Address of PCONSOLE Table--see Figure 7-12
BUFSIZ	Size of BUFFER. If this is less than the size of the PCONSOLE Table, only complete fields should be filled in.

GET must provide information for the FlexOS PCONSOLE Table. The PCONSOLE Table describes a physical console device and contains the information described in Table 7-11, below.

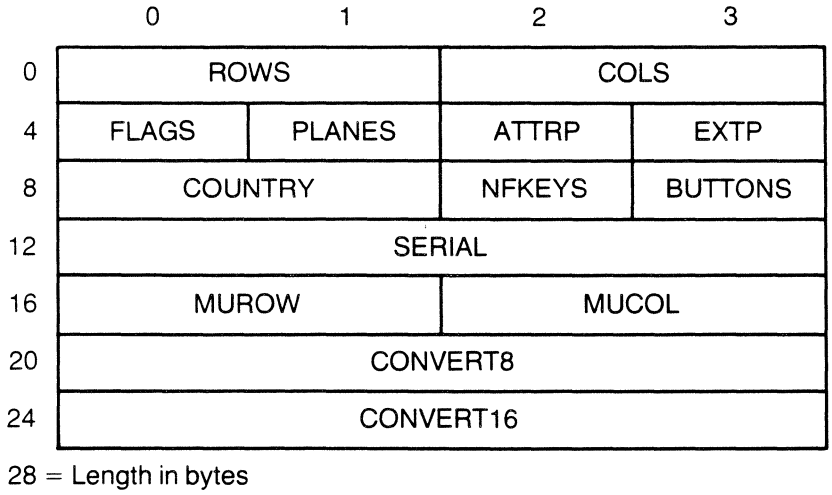


Figure 7-12. PCONSOLE Table

GET provides routines to translate the 8-bit output character set to the 16-bit output character set and the 16-bit input character set to the 8-bit input character set. If the driver does not provide these routines, the Console RM uses the standard conversion routines when interfacing with a given unit. The FlexOS standard input and output character sets and escape sequences are defined in Appendix A. Country codes are listed in Appendix C of the FlexOS Programmer's Guide.

For the use of computers in Japan, GET might provide a routine to translate 8-bit SHIFT-JIS characters to 16-bit SHIFT-JIS characters as defined by Digital Research/Japan.

Table 7-11. Fields in PCONSOLE Table

Field	Description
ROWS	Number of rows on physical console
COLS	Number of columns on physical console
FLAGS	Bit map of capabilities: Bit 0: 1 = graphic and character-mapped display 0 = character-only display Bit 1: 1 = no numeric keypad 0 = numerical keypad Bit 2: Reserved Bit 3: 1 = color screen 0 = monochrome screen Bit 4: 1 = memory-mapped video 0 = serial device Bit 5: 1 = currently in graphics mode 0 = currently in character mode
PLANES	Planes supported on PFRAME. FlexOS assumes whatever planes are supported on PFRAME are supported on VFRAME. Bit 0: 1 = character plane supported Bit 1: 1 = attribute plane supported Bit 2: 1 = extension plane supported Bits 3-7: Set to zero
ATTRP	Attribute plane bits supported
EXTP	Extension plane bits supported
COUNTRY	Country code
NFKEYS	Number of function keys

Table 7-11. (Continued)

Field	Description
BUTTONS	Number of mouse buttons
SERIAL	Serial number of mouse
MUROW	Number of mickey units per row
MUCOL	Number of mickey units per column
CONVERT8	Address of 8-bit to 16-bit output conversion routine. If NULLPTR, the FlexOS standard conversion routine is called internally. See the sample driver code for the specific interface expected of this routine.
CONVERT16	Address of 16-bit to 8-bit input conversion routine. If NULLPTR, the FlexOS standard conversion routine is called internally. See the sample driver code for the specific interface expected of this routine.

7.4.7 SET--Change the PCONSOLE Table

Parameter: Address of SET parameter block.

Return Code:

E_SUCCESS	Operation was successful
IOERROR	Driver-specific error code

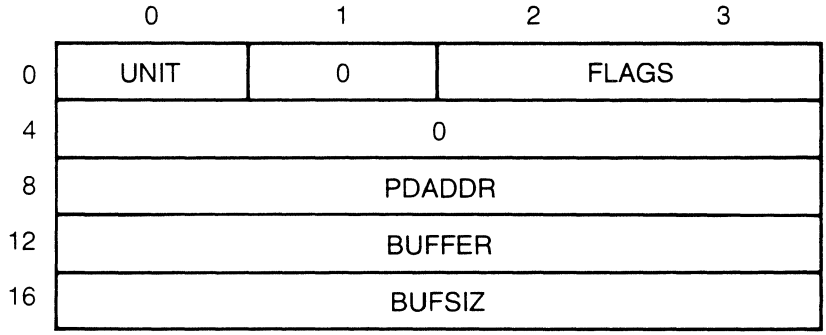


Figure 7-13. SET Parameter Block

Table 7-12. Fields in SET Parameter Block

Field	Description
UNIT	Driver unit number
0	Byte set to zero
FLAGS	Bit map of flag usage Bits 0-14: Reserved Bit 15: 1 = Buffer in User Memory 0 = Buffer in System Memory
0	Long word set to zero
PDADDR	Address of process descriptor of process containing BUFFER
BUFFER	Address of PCONSOLE Table

SET changes the COUNTRY field in the PCONSOLE Table. (See Table 7-11.) This is the only PCONSOLE Table value that can be modified. Support for this console function is optional.

End of Section 7

Disk Drivers

This section describes how FlexOS performs reads and writes to disk and defines I/O functions for disk drivers.

8.1 Disk Driver Input/Output

FlexOS supports an extended PC DOS 2.0 disk file format. The file system primitives are contained in the Disk Resource Manager (RM). The Disk RM manages the disk file system through the interface with the disk driver(s).

All hardware-dependent code is within the disk drivers. The Disk Resource Manager deals with a single, uniform disk driver interface. All types of disk media are handled through this single interface.

FlexOS supports three extensions to the PC DOS 2.0 disk file format. The first two extensions are required for file security, the third to support variable record sizes on files. These extensions take the form of fields in each directory entry that specify a file's User/Group ID, protection level, and record size. The FlexOS file system is described in detail in the FlexOS Programmer's Guide.

With minor modifications, the FORMAT and FDISK utilities, distributed in source code, support non-DOS disk formats. COPYCPM, also distributed in source code, allows you to copy to and from CP/M media.

8.1.1 Reentrancy at the Driver/Disk Controller Level

Disk drivers are organized at the controller level and the unit level. Each disk driver controls one disk controller and as many units as are controlled by that controller. Each unit represents a logically separate disk drive. A unit might be a single diskette drive or one partition of a partitioned hard disk.

The Disk Resource Manager supports reentrancy only at the controller/driver level. The unit level is always synchronized. A disk

driver can choose to allow only one operation at a time for all units of the driver or it can allow each unit to operate independently of the other. In either event, there can be only one operation at a time at the unit level.

In the FLAGS field of a disk driver's driver header, bit 0 can be 0 if the controller can handle multiple I/O requests. However, bit 1 must be 1, indicating that the disk driver synchronizes I/O requests at the unit level. See Section 4.2 for a complete description of the driver header.

8.1.2 Disk Driver Types

The Disk Resource Manager supports three types of disk drivers: removeable with open door support, removeable, and permanent. The Disk Resource Manager deals with each of these types differently to take advantage of each type's capabilities.

With all disk driver types, FlexOS allows delayed READs and WRITEs for those drivers performing intermediate buffering of data on I/O operations. The READ and WRITE disk driver functions have "normal read (or write)" and "read (or write) through" options. Normal reads and writes can take advantage of buffering, if implemented at the driver level. The read or write through option forces a direct read from or write to the actual medium, bypassing any intermediate buffering.

Delayed WRITEs are forced out to disk when FlexOS performs a CLOSE operation on a unit and on WRITE through and READ through operations. I/O will always occur in response to any of these three operations.

The disk drivers shipped with FlexOS do not use intermediate buffering.

Disk driver units inform the Disk RM of their driver type, size, and file structure information through a data structure called the Media Descriptor Block. The driver returns the Media Descriptor Block through the SELECT entry point. The Media Descriptor Block is described in detail in the explanation of SELECT, later in this section.

Removeable with Open Door Support

With disk drive hardware providing "open door" detection, the Disk Resource Manager ensures the integrity of removeable media.

To take advantage of the FlexOS open-door support, the disk driver must be able to respond to the hardware's open door interrupt. The driver responds to such an interrupt by setting an open door flag. The address of the open door flag is given to the Disk Resource Manager through a disk driver's GET entry point at the time a driver unit is initialized.

At each I/O operation, the Disk Resource Manager checks the address of the open door flag for a non-zero value. If it finds a non-zero value, the Disk RM requires verification that the disk has not been changed before passing the next I/O request. If a change is detected, the Disk RM calls the SELECT function to reinitialize the driver unit. Any intermediate buffers are not written to the disk.

The Disk RM does not do media verification at any time other than in response to the open door flag, thereby improving performance significantly over hardware and software without open door support.

Removable Without Open Door Support

While FlexOS obtains optimum performance from floppy disk hardware and software supporting an open door interrupt, FlexOS also supports disk drivers that do not have such an interrupt. For drives in this category, the Disk RM maintains checksum information on critical portions of the system area of the disk medium. If the drive is not used within a certain time interval, the volume is marked as suspect. At the next disk access, the checksum is verified. If the verification fails, the Disk RM calls the SELECT function to allow the disk driver unit to specify which type of media is in place.

After a failed verification, the Disk RM takes the following steps:

- The Disk RM assumes that the disk has been changed and disregards all buffers pertaining to the drive.

- If there are opened files on the removed medium, the Disk RM closes the files without flushing any intermediate buffers to the disk.
- The Disk RM attempts to re-login the disk. Then, if the SVC request did not assume an open file, the Disk RM retries the request. LOOKUP, OPEN, and RENAME are examples of SVCs that do not assume open files.
- The Disk RM sets the open door flag (see previous subsection) and returns an error code to the process that requested I/O.

To improve performance, FlexOS does not perform checksum verification on READs and WRITEs. This means that, if a file is active when a disk is changed, FlexOS could write data from that file to a changed disk. Digital Research recommends that you implement an open door interrupt to eliminate this possibility and to significantly enhance floppy disk performance.

Permanent

In this usage, "permanent" means that you cannot change the medium during the life of the system. The Disk RM does no checksum verification and does not check the open door flag. Because of these facts, I/O system performance is faster with permanent media than with either sort of removable media.

8.2 Logical Disk Layouts

This section illustrates a generalized logical disk layout and a logical layout for hard disks.

Each disk driver unit interfaces to a logical disk drive with the layout shown in Figure 8-1; Table 8-1 explains each field.

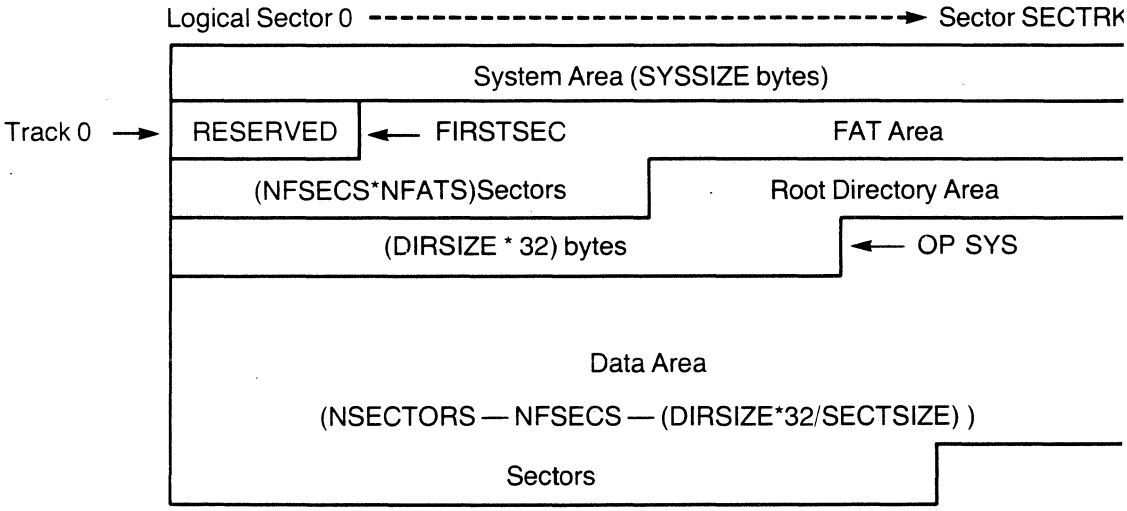


Figure 8-1. Logical Disk Layout

Table 8-1. Fields in Logical Disk Layout

Field	Description
Logical Sector 0	The first sector of the track containing the beginning of the FAT (File Allocation Table) area. This sector is also the sector identified by head 0, track 0, sector 0. By definition, track 0 contains the first sector of the FAT area. If the FAT area does not exist (NFSECS and NFATS are both zero), then track 0 is the track containing the first sector of the root directory. The Disk RM handles both one- and zero-based sector numbering. Programs in User Memory and disk drivers must have the same numbering scheme.
SECTRK	Specifies the number of physical sectors per track.
System Area	Usually defined as the area used for booting. The system area is considered outside of the disk medium and can be formatted independently of the disk medium. For hard disks, the system area is zero length and is therefore not counted in sequential sector numbering. For a disk containing tracks with different densities, the system area must end on a track boundary as defined for that disk. For a disk with a uniform density, if the system area extends into the beginning of the first track of the disk, it must end on a physical sector boundary, as defined for that disk.

Table 8-1. (Continued)

Field	Description
SYSSIZE	Size of the system area, in bytes.
RESERVED	Normally, when the system area is zero, this field contains the boot sector. At offset 0 in the boot sector is the BIOS Parameter Block, illustrated in Figure 8-4, below.
FIRSTSEC	The physical sector number of the first FAT sector on track 0. If no FAT exists, FIRSTSEC is the first sector of the root directory area.
NFSECS	The number of sectors in each FAT. NFSECS is zero for CP/M media.
NFATS	The number of FATs in the FAT area. NFATS is zero for CP/M media.
DIRSIZE	The number of root directory entries. A directory entry is 32 bytes long. The physical sectors occupied by the directory area must be contiguous.
NSECTORS	Specifies the total number of sectors on the disk. See below for the formula for determining NSECTORS.
SECTSIZE	Specifies the physical sector size of the disk, in bytes. Legal sizes are 128, 256, 512, 1024, 2048 and 4096.

The formula for determining the total number of sectors on a disk (NSECTORS) is as follows:

```
NSECTORS = FIRSTSEC + (NFATS*NFSECS) +
           (DIRSIZE * 32) + (SECSIZE - 1))/SECSIZE +
           (Number of Clusters * (Sectors per Cluster))
```

A cluster is the number of physical sectors per file allocation unit on a given disk.

If a disk is bootable, the operating system must be stored starting with first sector following the directory area.

Model Hard Disk Layout

Figure 8-2 illustrates a model logical hard disk. The following table describes the components.

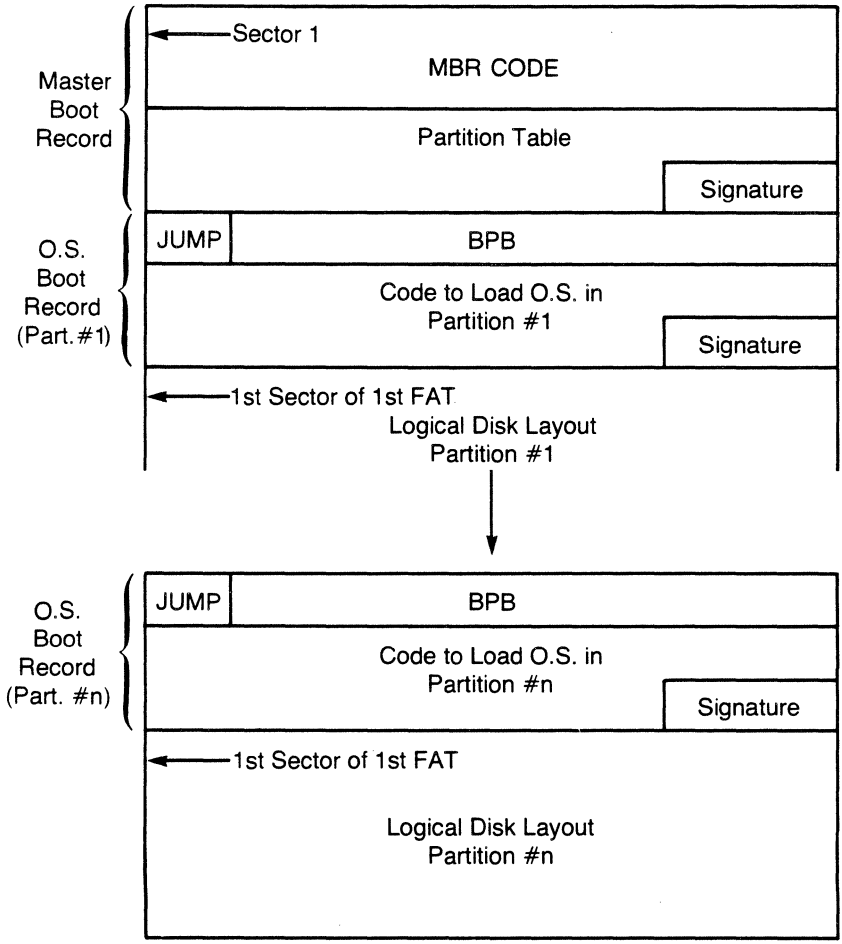


Figure 8-2. Hard Disk Layout

Table 8-2. Fields in Hard Disk Layout

Field	Description
Master Boot Record (MBR)	Contains code to load and pass control to the boot record for one of four possible operating systems. Also contains the Partition Table. For hard disks with a sector size of 512 bytes, the MBR is usually one sector long.
MBR_CODE	Code portion of Master Boot Record.
Partition Table	Contains information on each of four possible partitions on the hard disk. See Figure 8-3, below.
SIGNATURE	Two-byte field at offset 1FEH from the beginning of the MBR. A value of 55AAH indicates a valid partition.
O.S. Boot Record	Contains code and data to load an operating system. For hard disks with a sector size of 512 bytes, the O.S. Boot Record is usually one sector long.
JUMP	Instruction to pass control to boot loader code after ROM monitor reads boot record into memory.
BPB	BIOS Parameter Block. Table describing a given operating system's partition. See Figure 8-4, below.
Code to Load O.S.	Code portion of an operating system's boot record.
Logical Disk Layout	The LDL as it is illustrated in Figure 8-1, above.

The Partition Table is a structure beginning at offset 1BEH from the beginning of the Master Boot Record. Figure 8-3 illustrates its format.

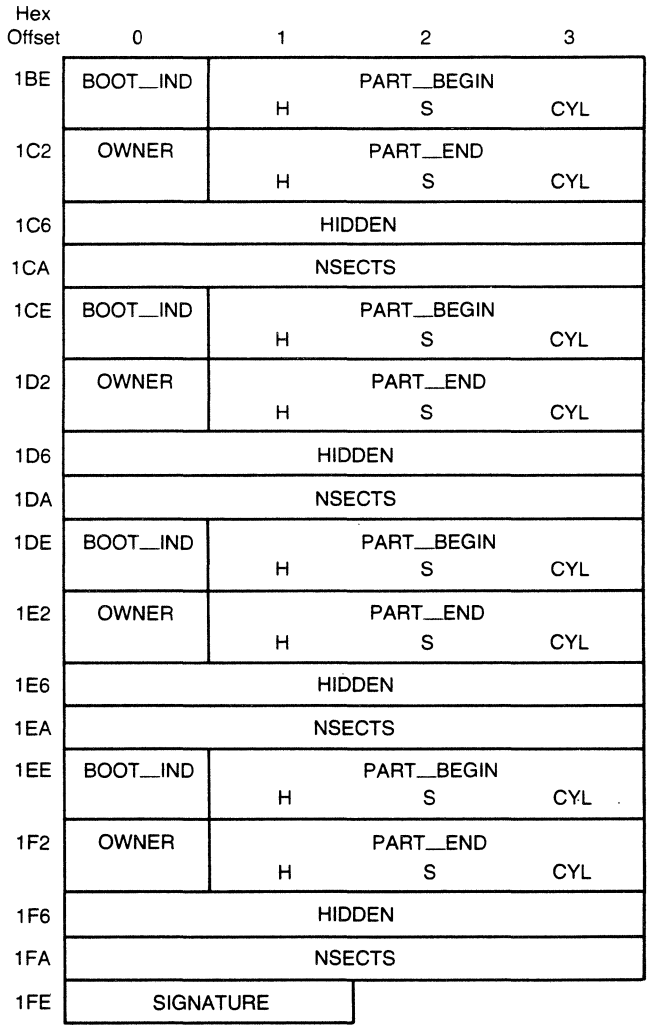


Figure 8-3. Partition Table

Table 8-3. Fields in Partition Table

Field	Description
BOOT_IND	Indicates whether a partition is bootable, where 0 indicates non-bootable and 80H indicates a bootable partition. Only one partition can be marked as bootable.
PART_BEGIN	Three-byte field indicating the head (H), sector (S), and cylinder (CYL) number where a partition begins. The head number is stored in the H field. The sector number is stored in the low order 6 bits of the S field. The cylinder number is 10 bits; the low order eight bits are stored in the CYL field, while the high order two bits are stored in the high order two bits of the S field. All partitions are usually allocated on track boundaries and begin on sector 1, head 0.
OWNER	One byte field indicates which operating system owns the partition. This field can contain one of the following values: 00H = unknown 01H = DOS 12-bit FAT entries 04H = DOS 16-bit FAT entries
PART_END	Three-byte field indicating the head (H), sector (S), and cylinder (CYL) numbers where a partition ends. See PART_BEGIN, above, for an explanation of how these values are stored.
HIDDEN	Four-byte field contains the number of sectors preceding a partition. Count sectors starting with cylinder 0, sector 1, head 0, incrementing the sector number up to the beginning of a partition. Store this value least significant word first.

Table 8-3. (Continued)

Field	Description
NSECTS	Number of sectors allocated to a partition. Store this four-byte value least significant word first.
SIGNATURE	Two-byte field at offset 1FEH from the beginning of the MBR. A value of 55AAH, stored high order byte first, indicates a valid partition.

The BIOS Parameter Block (BPB) is stored at offset 0 in an operating system's boot record. Each partition must contain a BPB, even if it is not bootable. Figure 8-4 illustrates the format of a BPB.

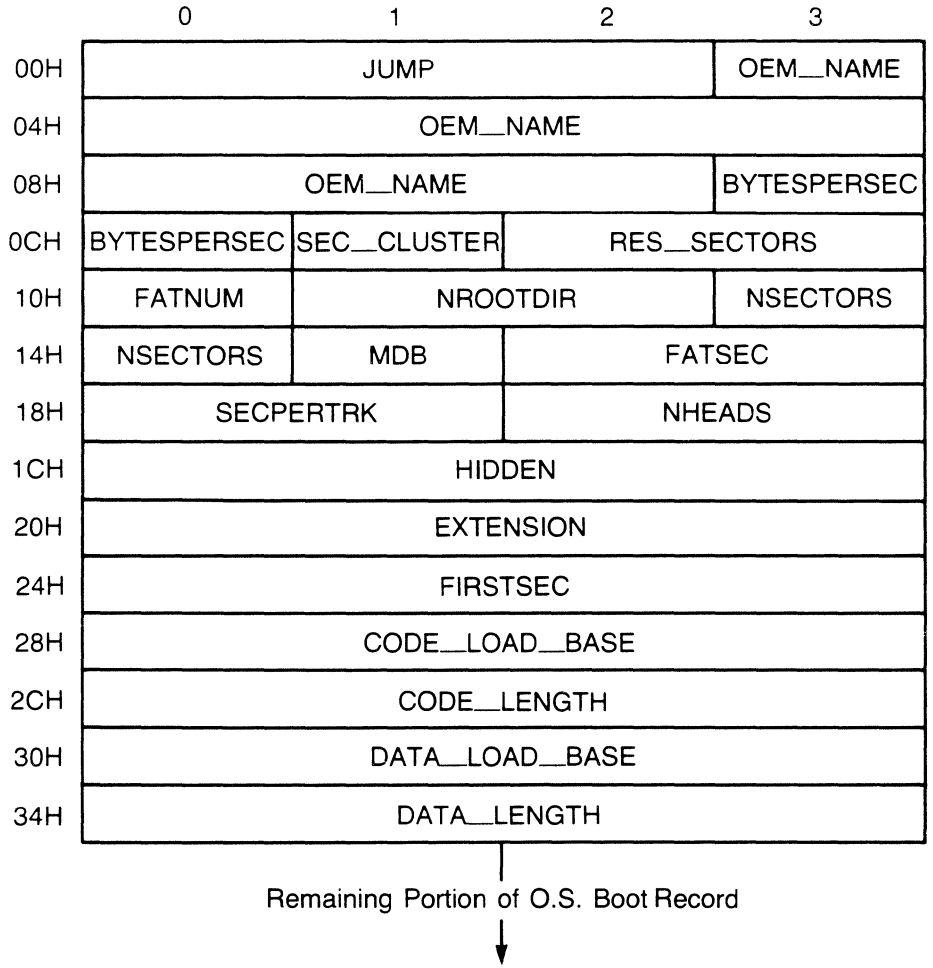


Figure 8-4. BIOS Parameter Block

Table 8-4. Fields in BPB

Field	Description
JUMP	A jump instruction to transfer control to an operating system's loader. See the chip-specific supplements for a description of the jump instruction.
OEM_NAME	The OEM name and version number identifying the boot record's operating system.
BYTESPERSEC	Number of bytes per sector.
SEC_CLUSTER	Number of sectors per file allocation unit in a partition. This value must be a power of two.
RES_SECTOR	Number of sectors reserved by the operating system, starting at logical sector 0.
FATNUM	Number of FATs in a partition.
NROOTDIR	Maximum number of root directory entries in a partition.
NSECTORS	Total number of sectors in a partition, including boot, directory, and reserved sectors. If this field value is 0, the EXTENSION field contains the total.
MDB	Media Descriptor Byte. Describes disk characteristics; MDB values are listed in Table 8-5, below.
FATSEC	Number of sectors occupied by one FAT.
SECPERTRK	Number of sectors per track in a partition.
NHEADS	Number of heads in partition.
HIDDEN	Total number of sectors preceding a partition, including sectors occupied by the MBR.

Table 8-4. (Continued)

Field	Description
EXTENSION	If NSECTORS contains zero, EXTENSION contains the total number of sectors in a partition. The total is here when the partition's size is too big for recording in NSECTORS.
FIRSTSEC	First sector of data area.
CODE_LOAD_BASE	Address where the operating system code is to be loaded.
CODE_LENGTH	Length, in bytes, of code segment.
DATA_LOAD_BASE	Address where the operating system data is to be loaded.
DATA_LENGTH	Length, in bytes, of data segment.

From the EXTENSION field through the end of the BPB is Digital Research's extension to the standard DOS BPB. The FORMAT utility fills in the fields from BYTESPERSEC through FIRSTSEC. The code and data load addresses and segment lengths are filled in by the SYS utility.

The Media Descriptor Byte have the following values:

Table 8-5. Media Descriptor Byte Values

Value	Meaning
F8H	Hard disk
F9H	Double-sided, 15 sectors per track
FCH	Single-sided, 9 sectors per track
FDH	Double-sided, 9 sectors per track
FEH	Single-sided, 8 sectors per track
FFH	Double-sided, 8 sectors per track

Values F9H through FFH refer to 5 1/4-inch diskettes.

8.3 Error Handling

The method used by Disk RM in handling errors depends on the error code returned by the driver unit and the type of media.

All FlexOS function return codes are 32-bit values. If the value is a negative number, it represents an error code. Error codes in the range from -64 to -2 gigabytes are driver-specific error codes. FlexOS system-wide error codes are listed in Appendix B of the FlexOS Programmer's Guide.

Disk driver functions that return physical errors return the error code to the application process, allowing the application to inform the user of the problem.

When I/O operations to removable media without open door support return a timeout error, the Disk RM automatically sets the open door flag and returns the error to the calling process. At the time of the next operation, the Disk RM performs a check to ensure that the medium has not changed.

8.4 Disk Driver I/O Functions

The following section describes the functions called by the Disk Resource Manager through the entry points contained in the disk driver's Driver Header. Of these functions, READ and WRITE are expected to be asynchronous; the remaining functions are synchronous. The physical I/O within the READ and WRITE functions is performed by Interrupt Service Routines (ISRs) and Asynchronous Service Requests (ASRs). Examples are contained in the source code for the sample disk drivers on the FlexOS distribution diskettes.

8.4.1 SELECT--Initialize driver unit

Parameter: Address of SELECT parameter block

Return Code:

E_SUCCESS	Successful operation
E_UNITNO	Drive has been installed to allow partitions (see FLAGS field of the INSTALL SVC's parameter block in the FlexOS Programmer's Guide) but driver is unable to read partition.
E_READY	Door open on a removable medium
E_CRC	Cyclical Redundancy Check error
E_SEEK	Non-existent track or sector
E_SEC_NOTFOUND	Sector or record not found
E_MISADDR	Missing address mark
E_DKATTACH	Attachment failed to respond
E_READFAULT	Read error
E_GENERAL	Undetermined source of failure

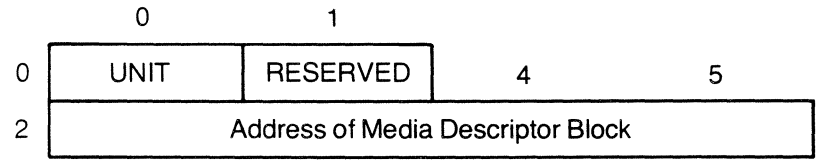


Figure 8-5. SELECT Parameter Block

In the figure above, UNIT refers to the driver unit number of a specific disk drive.

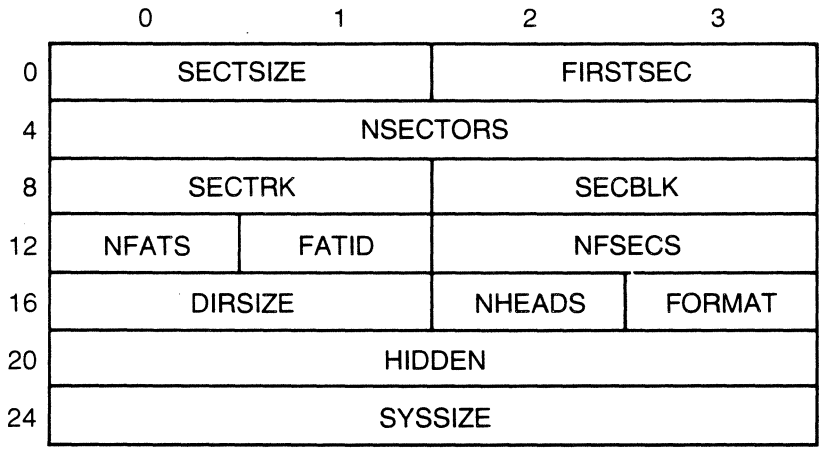


Figure 8-6. Media Descriptor Block

Table 8-6. Media Descriptor Block Fields

Field	Description
SECTSIZE	Physical sector size, in bytes. This value is required for a partial MDB.
FIRSTSEC	First physical sector number of FAT on track 0
NSECTORS	Number of sectors in logical disk image. This includes boot sector, FATs, directories, and data. The boot sector consists of a BIOS Parameter Block and code to load the operating system. Figure 8-4 illustrates the format of a BIOS Parameter Block. NSECTORS does not include system track(s). This value is required for a partial MDB.
SECTRK	Number of sectors per track
SECBLK	Number of sectors per block (file allocation unit)
NFATS	Number of FATs
FATID	Implementation-dependent value indicating media format
NFSECS	Number of physical sectors in a single FAT
DIRSIZE	Number of directory entries in the root directory
NHEADS	Number of heads
FORMAT	FAT format 0 = Raw 1 = 1 1/2-byte FATs 2 = 2-byte FATs For a partial MDB, FORMAT must be set to zero.

Table 8-6. Continued)

Field	Description
HIDDEN	Number of hidden sectors, that is, number of sequential physical sectors preceding a partition. HIDDEN is used only for partitioned disks. See the HIDDEN fields in the Partition Table (Figure 8-3) and BIOS Parameter Block (Figure 8-4).
SYSSIZE	Size of the system area of the disk, in bytes. The system area is outside of the disk medium and can be formatted independently of the disk medium. The system area might contain code to support an operating system other than FlexOS.

The Disk Resource Manager calls the SELECT function to initialize the driver unit for subsequent READ, WRITE, FLUSH and SPECIAL calls on the current medium. The Disk RM calls SELECT only once until either the "Media Change" error is detected or the drive has been opened exclusively.

SELECT is called with address of the SELECT parameter block. This parameter block contains the address of the Media Descriptor Block. The Media Descriptor Block determines the type (removable, permanent, and so forth) and size of the media as well as the file structure to be managed. It is a static structure and can be used for multiple units of the same driver if the MDB is identical for those units.

If SELECT is called for a unit containing an unformatted disk or a disk whose format is not supported by the Disk RM, the driver should not return an error code. Instead, the driver should return a partly filled-in MDB. By filling in the SECTSIZE, NHEADS, and FORMAT fields of the SELECT parameter block, the driver allows utilities, such as COPYCPM, to use the SPECIAL SVCs to initiate raw I/O to non-DOS media.

8.4.2 FLUSH--Flush intermediate buffers to media

Parameter: Address of FLUSH parameter block,

Return Code:

E_SUCCESS	Operation was successful
E_UNITNO	Invalid unit number
E_BADPB	Bad parameter block
E_READY	Door open on a removable medium
E_SEC_NOTFOUND	Sector or record not found
E_MISADDR	Missing address mark
E_SEEK	Non-existent track or sector
E_DKATTACH	Attachment failed to respond
E_WPROT	Disk write-protected
E_WRITEFAULT	Write error
E_GENERAL	Failure from undetermined source

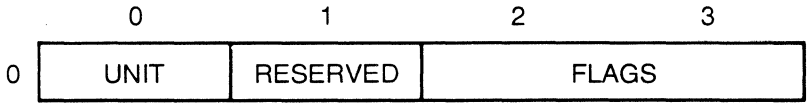


Figure 8-7. FLUSH Parameter Block

The UNIT value indicates the driver unit number of the drive to be flushed. The FLAGS field is reserved for future use.

The Disk Resource Manager calls the FLUSH function to flush any intermediate buffers to a medium and to make sure the driver is not in any intermediate state.

The disk drivers provided with FlexOS do not use intermediate buffering. When the Disk RM calls FLUSH in a FlexOS disk driver, FLUSH returns E_SUCCESS.

8.4.3 READ--Obtain data from disk medium

Parameter: Address of READ parameter block

Return Code:

emask	The return code from FLAGEVENT

The read event's completion code is returned through the FLAGSET function and should be one of the following:

E_SUCCESS	Successful operation
E_UNITNO	Drive has been installed to allow partitions (see FLAGS field of the INSTALL SVC's parameter block in the FlexOS Programmer's Guide) but driver is unable to read partition.
E_READY	Door open on a removable medium
E_CRC	Cyclical Redundancy Check error
E_SEEK	Non-existent track or sector
E_SEC_NOTFOUND	Sector or record not found
E_MISADDR	Missing address mark
E_DKATTACH	Attachment failed to respond
E_READFAULT	Write error
E_GENERAL	Failure from undetermined source

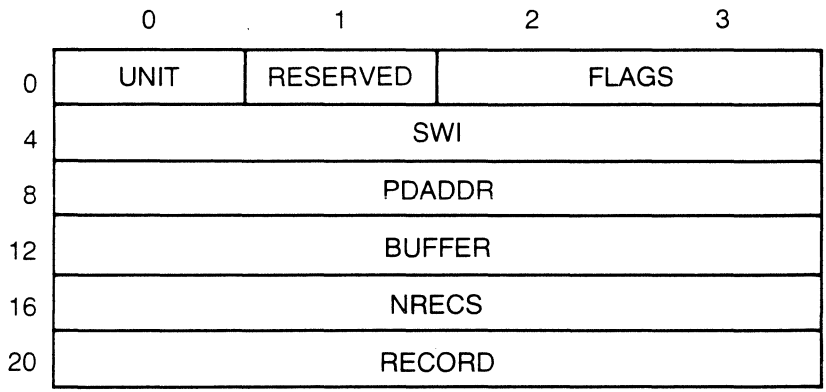


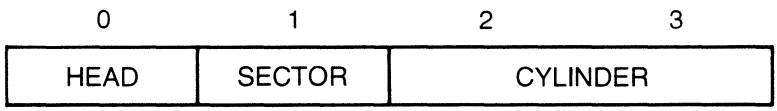
Figure 8-8. READ Parameter Block

Table 8-7. READ Parameter Block Fields

Field	Description
UNIT	Driver unit number
FLAGS	Bit map of flags
	Bit 0: 1 = Read through 0 = Normal read Read through option forces direct read from the medium, bypassing intermediate buffers. This flag is meaningless for drivers without intermediate buffers.
	Bit 1: 1 = RECORD formatted as head, sector, cylinder 0 = RECORD formatted as the logical sector number from the beginning of the disk medium.
	Bit 2: 1 = Verify medium, do not read 0 = Read
	Bit 8: 1 = Write 0 = Read
	Bits 9-10: 0 = Not Applicable 1 = FAT 2 = DIR 3 = Data
	Bit 11-14: Reserved
	Bit 15: 1 = User Address 0 = System Address

Table 8-7. (Continued)

Field	Description
SWI	User-supplied software interrupt to be passed as a parameter to the FLAGEVENT driver service function.
PDADDR	Process descriptor address of process calling READ SVC. If an address is specified and it is a User Address, this is the pdaddr you use for the MAPU driver service. This is not necessarily the process calling this entry point, and therefore not the pdaddr used in the FLAGSET function. The pdaddr obtained before calling FLAGEVENT is found via the driver header's Ready List Root (RLR) address.
BUFFER	Address of buffer to place information into.
NRECS	Number of physical sectors to READ
RECORD	First sector to READ. This field is either a logical sector number or a head, track, sector specification, depending on the value in bit 1 of the FLAGS field. A logical sector number is the number of physical sectors from the beginning of the disk medium, where Sector 0 is Track 0, Head 0, Sector 0. If bit 1 of the FLAGS field is set, the RECORD parameter is formatted as follows:



The Disk Resource Manager calls the READ function to obtain data from the disk medium. The Disk RM assumes this function is asynchronous.

To work asynchronously, the READ driver function must call the FLAGEVENT driver service function to receive an event mask, which is returned to the Disk Resource Manager. Upon completion of the READ, FLAGSET is called by the asynchronous portion of the driver to return a completion code. FLAGEVENT and FLAGSET are explained in Section 5.1, "Flag System."

8.4.4 WRITE--Write data to disk medium

Parameter: Address of WRITE Parameter Block

Return Code:

emask	Return code from the FLAGEVENT driver service

The write events completion code is returned through the FLAGSET function and should be one of the following:

E_SUCCESS	Successful operation
E_UNITNO	Invalid unit number
E_BADPB	Bad parameter block
E_READY	Door open on a removable medium
E_SEC_NOTFOUND	Sector or record not found
E_MISADDR	Missing address mark
E_SEEK	Non-existent track or sector
E_DKATTACH	Attachment failed to respond
E_WPROT	Disk write-protected
E_WRITEFAULT	Write error
E_GENERAL	Failure from undetermined source

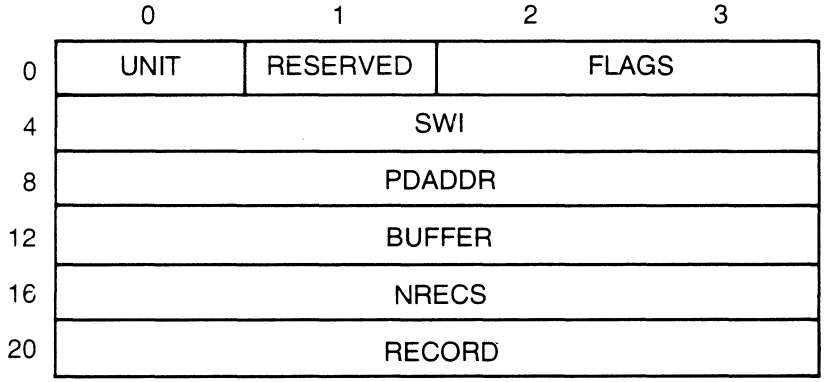


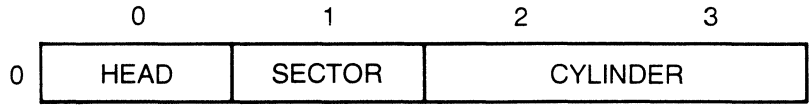
Figure 8-9. WRITE Parameter Block

Table 8-8. WRITE Parameter Block Fields

Field	Description
UNIT	Driver unit number
FLAGS	Bit map of flags
	Bit 0: 1 = Write through 0 = normal write The write through option forces a direct write to the actual media, bypassing any intermediate buffers. This flag has no meaning for disk drivers not using intermediate buffering.
	Bit 1: 1 = RECORD is formatted as head, track, cylinder 0 = RECORD is formatted as the logical sector number from the beginning of the disk medium.
	Bit 8: 1 = Write 0 = Read
	Bits 9-10: 0 = Not Applicable 1 = FAT 2 = DIR 3 = Data
	Bits 11-14: Reserved
	Bit 15: 1 = User Address 0 = System Address

Table 8-8. (Continued)

Field	Description
SWI	User-supplied software interrupt to be passed as a parameter to the FLAGEVENT driver service function.
PDADDR	Process descriptor address of process initiating the WRITE request. If an address is specified and the address is a User Address, this is the pdaddr you use for the MAPU driver service function. This is not necessarily the process calling this entry point and therefore not the pdaddr used in the FLAGSET function. The pdaddr obtained before calling FLAGEVENT is found through the Ready List Root (RLR) address in the driver header.
BUFFER	Address of buffer to obtain data from
NRECS	Number of physical sectors to WRITE
RECORD	First sector to WRITE. This field is either a logical sector number or a head, sector, cylinder specification, depending on the value in bit 1 of the FLAGS field. A logical sector number is the number of physical sectors from the beginning of the disk medium, where sector 0 is track 0, head 0, sector 0. If bit 1 of the FLAGS field is set, the RECORD parameter is formatted as follows:



The Disk Resource Manager calls WRITE to place data onto the disk medium. WRITE is assumed to be asynchronous.

The WRITE driver function must call the FLAGEVENT driver service to receive an event mask, which is returned to the Disk Resource Manager. Upon completion of the WRITE, the asynchronous portion of the driver calls the FLAGSET driver service to return a completion code. WRITE should then call the RETURN SVC through SUPIF to clear the event from the system.

8.4.5 SPECIAL Entry Point

The Disk Resource Manager calls the SPECIAL entry point to perform actions that cannot be performed by other disk driver functions. FlexOS defines six SPECIAL disk driver functions, 0 through 3, 8, and 9. The Disk RM reserves functions 10-31 for future use. Functions 32-63 are OEM-dependent and can be used for special activities particular to a given hardware implementation.

SPECIAL Function 0--Read from System Area of disk

Parameter: Address of SPECIAL parameter block

Return Code:

E_SUCCESS	Successful operation
E_UNITNO	Invalid unit number
E_READY	Door open on a removable medium
E_CRC	Cyclical Redundancy Check error
E_SEEK	Non-existent track or sector
E_SEC_NOTFOUND	Sector or record not found
E_MISADDR	Missing address mark
E_DKATTACH	Attachment failed to respond
E_READFAULT	Write error
E_GENERAL	Failure from undetermined source

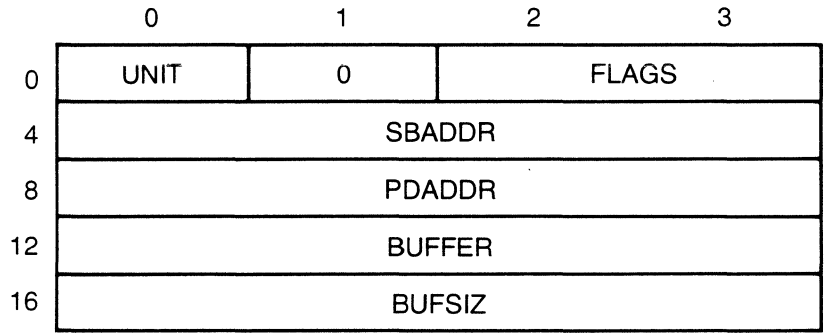


Figure 8-10. SPECIAL Function 0 Parameter Block

Table 8-9. SPECIAL Function 0 Parameter Block Fields

Field	Description
UNIT	Driver unit number
0	SPECIAL function number
FLAGS	Bit map of flags Bits 0-13: Driver-type specific Bit 14: Reserved Bit 15: 1 = User Address 0 = System Address
SBADDR	System address of special buffer for blocking/deblocking. The drivers shipped with FlexOS do not use blocking/deblocking.
PDADDR	Process descriptor address of process initiating the SPECIAL request. If an address is specified and the address is a User Address, this is the pdaddr you use for the MAPU driver service function. This is not necessarily the process calling this entry point and therefore not the pdaddr used in the FLAGSET function. The pdaddr obtained before calling FLAGEVENT is found through the Ready List Root (RLR) address in the driver header.
BUFFER	Address of buffer where data will be placed.
BUFSIZ	Size of buffer, in bytes

SPECIAL function 0 reads the data in the system area of the disk and places the data into the specified buffer. This function is performed synchronously and does not return until the read is complete.

SPECIAL Function 1--Write to System Area of disk

Parameter: Address of SPECIAL parameter block

Return Code:

E_SUCCESS	Successful operation
E_UNITNO	Invalid unit number
E_BADPB	Bad parameter block
E_READY	Door open on a removable medium
E_SEC_NOTFOUND	Sector or record not found
E_MISADDR	Missing address mark
E_SEEK	Non-existent track or sector
E_DKATTACH	Attachment failed to respond
E_WPROT	Disk write-protected
E_WRITEFAULT	Write error
E_GENERAL	Failure from undetermined source

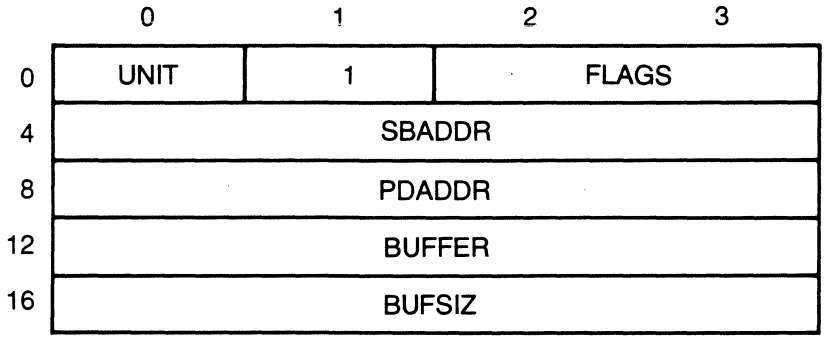


Figure 8-11. SPECIAL Function 1 Parameter Block

Table 8-10. SPECIAL Function 1 Parameter Block Fields

Field	Description
UNIT	Driver unit number
41	SPECIAL function number (in hex)
FLAGS	Bit map of flags Bits 0-13: Driver-type specific Bit 14: Reserved Bit 15: 1 = User Address 0 = System Address
SBADDR	System Address of special buffer for blocking/deblocking. The disk drivers shipped with FlexOS do not use blocking/deblocking.
PDADDR	Process descriptor address of process initiating the SPECIAL request. If an address is specified and the address is a User Address, this is the pdaddr you use for the MAPU driver service function. This is not necessarily the process calling this entry point and therefore not the pdaddr used in the FLAGSET function. The pdaddr obtained before calling FLAGEVENT is found through the Ready List Root (RLR) address in the driver header.
BUFFER	Address of buffer from which data will be written.
BUFSIZ	Size of buffer, in bytes

SPECIAL function 1 writes the data in the specified buffer to the system area of the disk. This function is performed synchronously and does not return until the write is complete.

SPECIAL Function 2--Format System Area of disk

Parameter: Address of SPECIAL parameter block

Return Code:

E_SUCCESS	Successful operation
E_UNITNO	Invalid unit number
E_BADPB	Bad parameter block
E_READY	Door open on a removable medium
E_SEC_NOTFOUND	Sector or record not found
E_MISADDR	Missing address mark
E_SEEK	Non-existent track or sector
E_DKATTACH	Attachment failed to respond
E_WPROT	Disk write-protected
E_WRITEFAULT	Write error
E_GENERAL	Failure from undetermined source

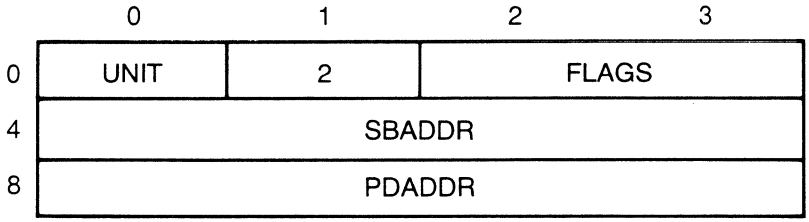


Figure 8-12. SPECIAL Function 2 Parameter Block

Table 8-11. SPECIAL Function 2 Parameter Block Fields

Field	Description
UNIT	Driver unit number
2	SPECIAL function number
FLAGS	Bit map of flags Bits 0-14: Reserved Bit 15: 1 = User Address 0 = System Address
WSBADDR	System address of buffer
PDADDR	Process descriptor address of process initiating the SPECIAL request. If an address is specified and the address is a User Address, this is the pdaddr you use for the MAPU driver service function. This is not necessarily the process calling this entry point and therefore not the pdaddr used in the FLAGSET function. The pdaddr obtained before calling FLAGEVENT is found through the Ready List Root (RLR) address in the driver header.

SPECIAL function 2 formats the system area of the disk managed by the specified driver unit. Function 2 only formats the system area if it resides in those tracks preceding the data area of the disk. In the FlexOS logical disk layout, the system area is not considered part of the disk medium and so can be formatted independently of the disk medium. This function is performed synchronously and does not return until the function is complete.

SPECIAL Function 3--Format track

Parameter: Address of SPECIAL parameter block

Return Code:

E_SUCCESS	Successful operation
E_UNITNO	Invalid unit number
E_BADPB	Bad parameter block
E_READY	Door open on a removable medium
E_SEC_NOTFOUND	Sector or record not found
E_MISADDR	Missing address mark
E_SEEK	Non-existent track or sector
E_DKATTACH	Attachment failed to respond
E_WPROT	Disk write-protected
E_WRITEFAULT	Write error
E_GENERAL	Failure from undetermined source

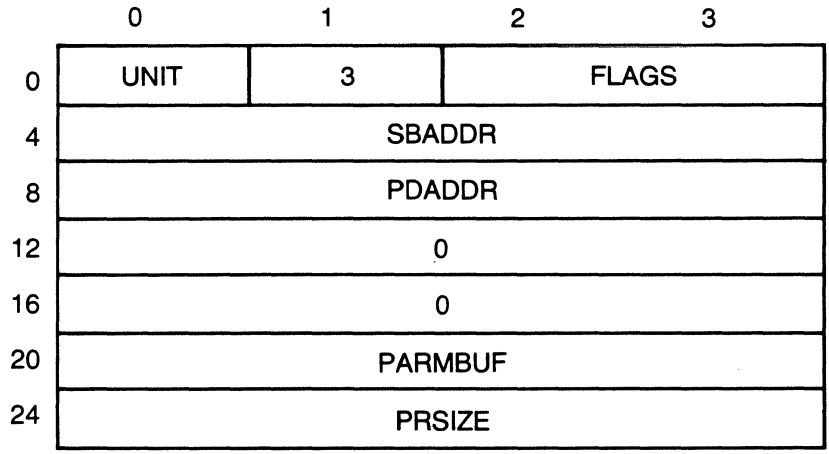


Figure 8-13. SPECIAL Function 3 Parameter Block

Table 8-12. SPECIAL Function 3 Parameter Block Fields

Field	Description
UNIT	Driver unit number
3	SPECIAL function number (in hex)
FLAGS	Bit map of flags:
	Bit 0: Reserved
	Bit 1: 0 = Track map is valid 1 = Map bad tracks
	Bit 2: 0 = Use HEAD, SECTOR, and BYTESPERSPEC fields. 1 = Ignore HEAD, SECTOR, and BYTESPERSEC fields. Instead, use a table of four-byte C-H-S-N fields as defined in the PARMBUF structure in Figure 8-13, below.
	Bit 3: 0 = SECTOR field is the starting sector number. This field is the first in variable-length list, whose length is the number of sectors per track. 1 = HEAD is the valid head number.
	Bits 4-14: Reserved
	Bit 15: 0 = System Address 1 = User Address

Table 8-12. (Continued)

Field	Description
SBADDR	System address of special buffer for blocking/deblocking. The drivers shipped with FlexOS do not use blocking/deblocking.
PDADDR	Process descriptor address of process initiating the SPECIAL request. If an address is specified and the address is a User Address, this is the pdaddr you use for the MAPU driver service function. This is not necessarily the process calling this entry point and therefore not the pdaddr used in the FLAGSET function. The pdaddr obtained before calling FLAGEVENT is found through the Ready List Root (RLR) address in the driver header.
PARMBUF	Address of data structure illustrated in Figure 8-13, below.
PRSIZE	Length, in bytes, of data buffer.

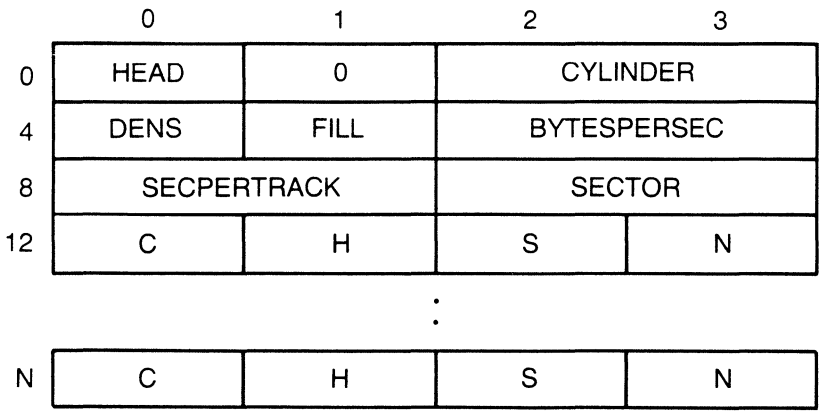


Figure 8-14. PARMBUF Structure

Table 8-13. PARMBUF Structure Fields

Field	Description
HEAD	If FLAGS bit 1 in SPECIAL Function 3 parameter block is zero, HEAD is a valid starting head number.
0	One byte set to zero
CYLINDER	Cylinder number
DENS	Density, where: 0 = single density 1 = double density
FILL	Fill character
BYTESPERSEC	If FLAGS bit 2 in SPECIAL Function 3 parameter block is zero, BYTESPERSEC is the number of bytes per sector.
SECPERTRACK	Number of sectors per track
SECTOR	This field's value depends on the settings for bits 2 and 3 in the FLAGS field of the SPECIAL Function 3 parameter block. If bits 2 and 3 are off, SECTOR contains the starting sector number. Use only bytes 0 through 11 of the PARMBUF Structure. If bit 2 is on and bit 3 is off, ignore SECTOR and use list starting with byte 12 in the PARMBUF Structure, as shown in Figure 8-13. If bit 2 is off and bit 3 is on, SECTOR is the first in variable-length list of sectors whose length is the number of sectors per track.
C-H-S-N	If FLAGS bit 2 in SPECIAL Function 3 parameter block is one, this is a variable-length list of four-byte fields, where C is cylinder, H is head, S is sector, and N is bytes per sector.

SPECIAL function 3 is used to format the disk medium and to map bad tracks out of the data area of a disk. SPECIAL function 3 maps bad tracks by marking a track in the FAT Table as allocated and without an owner. This function does not deal with the system area of a disk.

SPECIAL function 3 is called by the FORMAT utility.

SPECIAL Function 8--Initialize format

Parameter: Address of SPECIAL parameter block

Return Code:

E_SUCCESS	Successful operation
E_BADPB	Bad MDB parameters

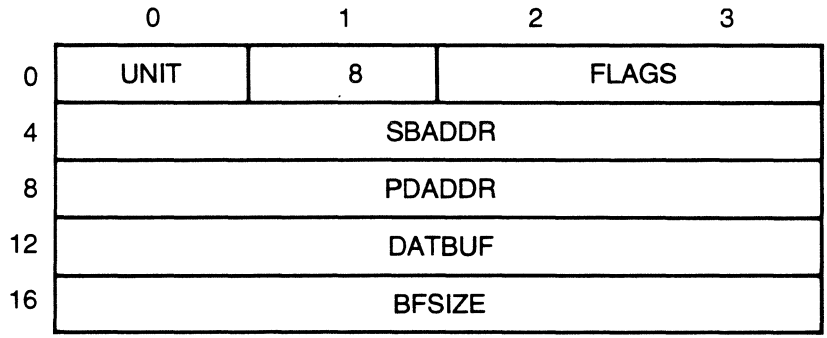


Figure 8-15. SPECIAL Function 8 Parameter Block

Table 8-14. SPECIAL Function 8 Parameter Block Fields

Field	Description
UNIT	Driver unit number
8	SPECIAL function number (in hex)
FLAGS	Bits 0-15 are reserved
SBADDR	System Address of special buffer
PDADDR	Process descriptor address of process initiating the SPECIAL request. If an address is specified and the address is a User Address, this is the pdaddr that must be used for the MAPU driver service function. This is not necessarily the process calling this entry point and therefore not the pdaddr used in the FLAGSET function. The pdaddr obtained before calling FLAGEVENT is found through the Ready List Root (RLR) address in the driver header. The RLR address is explained in Section 4, "Driver Interface."
DATBUF	Address of buffer containing Media Descriptor Block (MDB). The MDB is described under the SELECT function, in Figure 8-6 and Table 8-6, above.
BFSIZE	Size, in bytes, of DATBUF

SPECIAL function 8 resets the Media Descriptor Block in system and driver memory, but does not transfer the MDB information to the disk. This function enables a user program to begin formatting a disk by establishing a new set of guidelines for the disk. When formatting is complete, the MDB is written to the disk's system area.

SPECIAL Function 9--Get Drive Information

Parameter: Address of SPECIAL parameter block

Return Code:

E_SUCCESS	Successful operation
E_UNITNO	Invalid unit number
E_BADPB	Bad parameter block

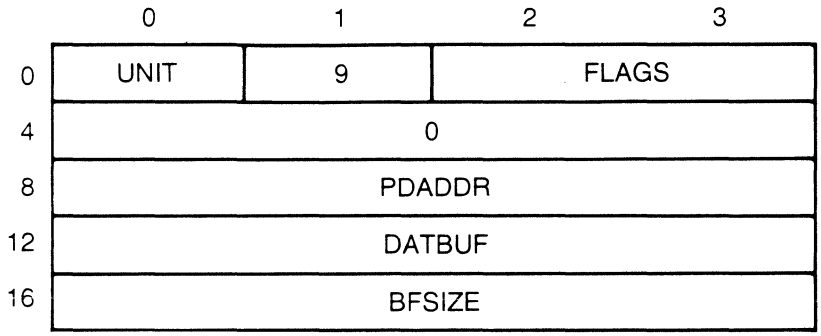


Figure 8-16. SPECIAL Function 9 Parameter Block

Table 8-15. SPECIAL Function 9 Parameter Block Fields

Field	Description
UNIT	Driver unit number
9	SPECIAL function number (in hex)
FLAGS	Bits 0-15 are reserved
PDADDR	Process descriptor address of process initiating the SPECIAL request. If an address is specified and the address is a User Address, this is the pdaddr that must be used for the MAPU driver service function. This is not necessarily the process calling this entry point and therefore not the pdaddr used in the FLAGSET function. The pdaddr obtained before calling FLAGEVENT is found through the Ready List Root (RLR) address in the driver header. The RLR address is explained in Section 4, "Driver Interface."
DATBUF	Address of buffer for Physical Unit Descriptor. (see Listing 8-1).
BFSIZE	Size, in bytes, of DATBUF

SPECIAL Function 9 requests disk-dependent information. You return the data in the buffer provided in the parameter block. Listing 8-1 describes the buffer contents.

Listing 8-1. SPECIAL Function 9 Physical Unit Descriptor

```
/*  PUD - Physical Unit Descriptor      */

PUD
{
    UWORD   pu_maxcyl ;      /*  max cyl number for i/o      */
    UWORD   pu_precomp ;     /*  precompensation cyl number  */
    UWORD   pu_crashpad ;    /*  landing zone cyl number     */
    UBYTE   pu_nheads ;      /*  no of heads                 */
    UBYTE   pu_sectors ;     /*  no sectors/track            */
    UBYTE   pu_step ;        /*  step rate                   */
    UBYTE   pu_eat ;         /*. even (unused)               */
} ;
```

8.4.6 GET--Provide unit-specific information

Parameter: Address of GET parameter block

Return Code:

E_SUCCESS	Successful operation
E_BADPB	Bad parameter block

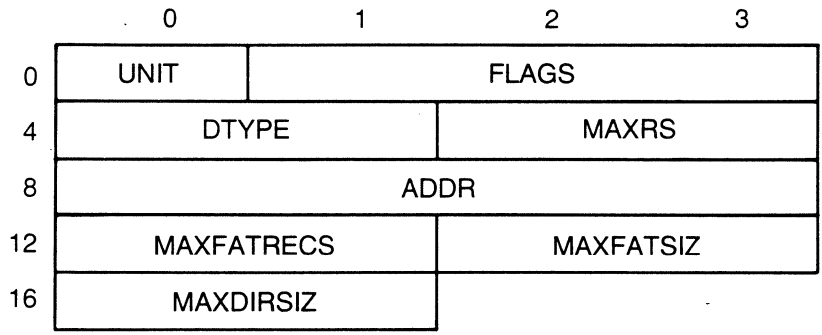


Figure 8-17. GET Parameter Block

Table 8-16. GET Parameter Block Fields

Field	Description
UNIT	Driver unit number
FLAGS	Reserved
DTYPE	Type of disk medium Bit 0: 1 = Removable media 0 = Permanent media Bit 1: 1 = Open door support 0 = No open door support Bit 2: Reserved
MAXRS	Maximum Record Size. This is the maximum physical sector size of all media types supported through this disk driver unit. For example, if this unit supports both single- and double-density diskettes, the larger of the physical sector sizes should be stated here. This field determines the size of the buffers the Disk Resource Manager maintains for the unit.
ADDR	Address of the open door byte if this is a disk drive with open-door-interrupt support.
MAXFATRCS	Maximum number of FAT records in a single FAT for all media types supported through this driver unit.
MAXFATSIZE	Maximum size of FAT, in bytes.
MAXDIRSIZE	Maximum number of root directory entries.

The Disk Resource Manager calls the GET function during the installation of the driver unit. GET is responsible for passing information to the Disk Resource Manager that is unit-specific, but does not pass the current disk medium-specific information.

The GET function passes the address of the GET parameter block to the driver unit and expects all of the fields of the parameter block except the UNIT and FLAGS fields to be filled in before returning.

8.4.7 SET--Change unit-specific information

Parameter: None

Return Code: E_IMPLEMENT Not Implemented

The Disk Resource Manager never calls the SET disk driver entry point. SET should return the "Not Implemented" error code.

End of Section 8

Port Drivers

This section describes the driver interface for interrupt-driven serial port drivers. All port drivers fall under the category of special drivers and are managed by the Miscellaneous Resource Manager.

Many serial interfaces generate interrupts only when a character is received, not when the port is ready to transmit a character. To account for this situation, the READ function in the sample port driver uses an ISR-ASR method of receiving characters, while the WRITE function uses the POLLEVENT driver service (see Section 5.3) to poll the selected port. Section 5, "Driver Services," discusses methods for responding to interrupts.

9.1 Port Driver Overview

A single port driver can control multiple units of the same type of port. FlexOS does not have a theoretical limit to the number of ports that are part of a system.

To allow multiple processes to perform serial I/O, the port driver should be I/O re-entrant at the driver and Resource Manager levels, and synchronized at the unit level. This means that bit 1 in the flags field of the port's Driver Header should be set. See Section 4.2, "Driver Header," for a definition of the Driver Header. See Section 11.1 for a discussion of how the Miscellaneous Resource Manager protects its drivers from user processes.

9.2 Port Driver I/O Functions

This section describes the port I/O functions accessed by the Miscellaneous Resource Manager through entry points in the port driver's Driver Header.

The port driver contains the SELECT, FLUSH, READ, WRITE, GET, and SET functions. The SPECIAL function is not required by the port driver and should return E_IMPLEMENT unless you provide support for SPECIAL calls.

See Section 4.4, "Driver Installation Functions," for a description of the INIT, SUBDRIVE, and UNINIT driver installation functions.

9.2.1 SELECT--Enable the specified unit

Parameter: Address of SELECT parameter block

Return Code:

E_SUCCESS Port is enabled

IO_ERROR Port not enabled

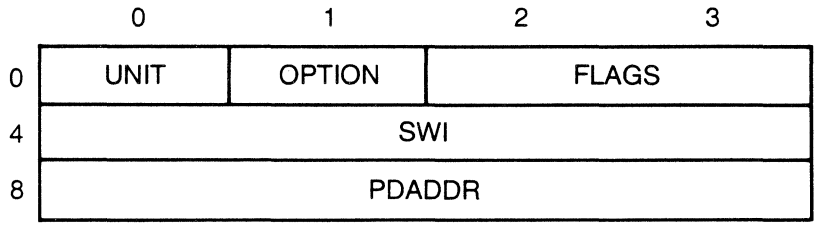


Figure 9-1. Port Driver SELECT Parameter Block

Table 9-1. Port Driver SELECT Parameter Block Fields

Field	Description
UNIT	Unit number of port being enabled
OPTION	User-defined option
FLAGS	User flags field
SWI	Address of optional Software Interrupt Routine. 0 if there is no SWI
PDADDR	Process descriptor address of process attempting to open this device (via the OPEN SVC)

The Miscellaneous RM calls SELECT to enable a specific port unit for I/O. SELECT clears all buffers for the selected unit, excluding the Interrupt Service Routine (ISR) buffer and then enables serial interrupts.

9.2.2 FLUSH--Disable port

Parameter: Address of FLUSH parameter block

Return Code:

E_SUCCESS	Port is deselected
IO_ERROR	Port not deselected

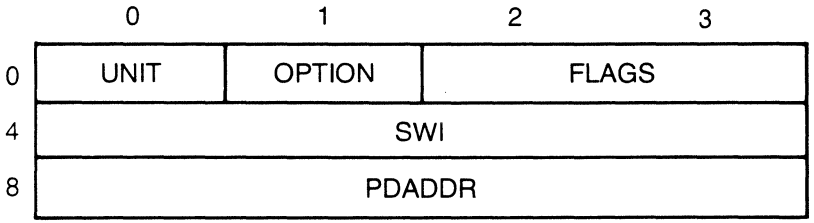


Figure 9-2. Port Driver FLUSH Parameter Block

Table 9-2. Port Driver in FLUSH Parameter Block Fields

Field	Description
UNIT	Unit number of port to be disabled
OPTION	User-defined option
FLAGS	User flags field
SWI	Address of optional Software Interrupt Routine. 0 if there is no SWI
PDADDR	Process descriptor address of process attempting to close this device (via the CLOSE SVC)

The Miscellaneous Resource Manager calls FLUSH before writing to another unit connected to the same driver or before uninstalling the driver.

I/O is not allowed past the point of invoking FLUSH without first calling SELECT. Because of this, FLUSH should clear any buffers not yet sent to the port, including the ISR buffer, before disabling serial interrupts.

9.2.3 READ--Read data from port

Parameter: Address of READ parameter block

Return Code:

emask	Event mask used by the calling process to wait for port to finish reading the character or characters into buffer
IO_ERROR	Unable to read from port

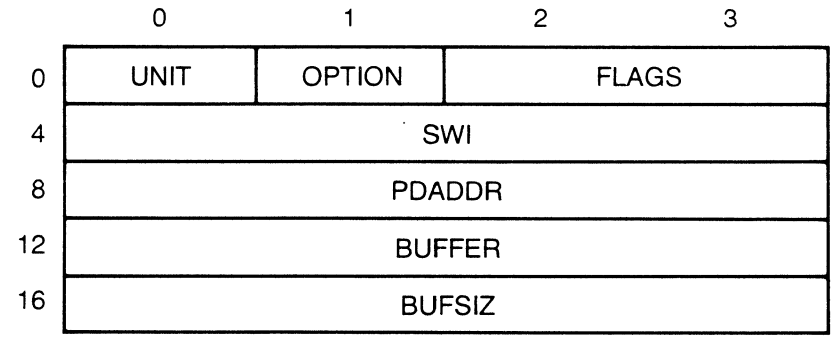


Figure 9-3. Port Driver READ Parameter Block

Table 9-3. Port Driver READ Parameter Block Fields

Field	Description
UNIT	Unit number of port being read
OPTION	User-defined option
FLAGS	User flags field
SWI	Address of optional Software Interrupt Routine; the value is 0 if there is no SWI
PDADDR	Process descriptor address of process attempting the read. This is not necessarily the process calling this entry point and therefore not the PDADDR used with FLAGEVENT and FLAGSET. Find the PDADDR of the process calling FLAGEVENT through the RLR field of the Driver Header. If an address is specified and it is a User Address, this is the PDADDR you use with MAPU.
BUFFER	Pointer to user's buffer
BUFSIZ	Size of buffer indicated in BUFFER

The Miscellaneous RM calls READ to read characters from a selected port. FlexOS assumes that the serial interface produces an interrupt when a character arrives. For those ports not interrupt-driven, see Section 5.3, "Device Polling."

READ must be able to buffer characters arriving at the port when the user process is not ready to read them. The READ function in FlexOS's sample serial driver works in the following sequence:

1. A character arrives at the serial port, causing an interrupt.
2. The operating system receives the interrupt via the exception vector established by the SETVEC driver service in the serial driver's INIT code. FlexOS passes control to the driver's Interrupt Service Routine (ISR).
3. The ISR reads the character from the serial port and calls the DOASR driver service to queue an ASR, passing DOASR the character as an argument.
4. The ASR puts the character into a buffer, then exits.

READ must transfer the characters from the ISR buffer into the user buffer. To do this, READ calls FLAGEVENT with the number of a clear flag and the address of the SWI in the READ parameter block. FLAGEVENT returns an event mask, which the driver saves.

The driver then calls DOASR with the address of the READ parameter block and the address of an ASR that performs the actual reading from ISR buffer to user buffer. When the READ ASR has completed, the driver calls FLAGSET to note the completion of the read.

9.2.4 WRITE--Send data to port

Parameter: Unsigned word (UWORD) with the unit number in the high order byte and the character in the low order word

Return Code:

emask	Event mask use by calling process to wait for port to be ready for more characters
IO_ERROR	Unable to write to port

The Miscellaneous RM calls WRITE to write the character provided to the specified unit.

If a port does not generate an interrupt when it is ready to transmit a character, WRITE can use the POLLEVENT driver service to poll. Alternatively, if the amount of data to write is small and/or the serial baud rate is fast, the driver can keep reading the status port until it becomes ready.

9.2.5 GET--Provide unit-specific information

Parameter: Address of GET parameter block

Return Code:

E_SUCCESS	Write to buffer completed - no error
E_xxx	Driver-specific error code

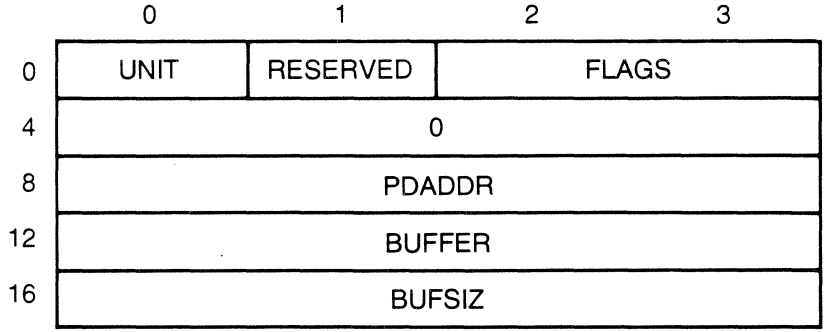


Figure 9-4. Port Driver GET Parameter Block

Table 9-4. Port Driver GET Parameter Block Fields

Field	Description
UNIT	Port driver unit number
FLAGS	Bit map of flags Bits 0-7: Defined by driver Bits 8-14: Reserved Bit 15: 0 = System Address 1 = User Address
PDADDR	Process descriptor address of the process that initiated the GET request. If the buffer address is a User Address, and the asynchronous portion of the driver is required to access the buffer, this parameter is used to call the MAPU driver service.
BUFFER	Address of buffer in which to write information from the driver's GET/SET Table; see Figure 9-5 below.
BUFSIZ	Size of buffer. This field determines the amount of information returned from the driver's GET/SET Table.

The Miscellaneous Resource Manager calls the GET entry point to place information from the port driver's GET/SET Table into a buffer whose address is specified as a parameter. The BUFSIZ parameter is passed to determine the amount of information to be obtained. If the buffer's size is less than the size of the table, only those fields that fit into the buffer are written there.

The GET and SET routines are not expected to return an event mask. The calling process should not return until the operation is complete. If the asynchronous portion of the driver is required to initiate an I/O event to obtain the information, the GET and SET routines must perform their own WAIT and RETURN SVCs through the Supervisor interface described in Section 6.

Many of the GET/SET Table values cannot be determined until they are placed in the table by SET. GET/SET Table values are set by one process for use by another.

The port driver GET/SET Table format is shown in Figure 9-5. Table 9-5 describes the GET/SET Table fields.

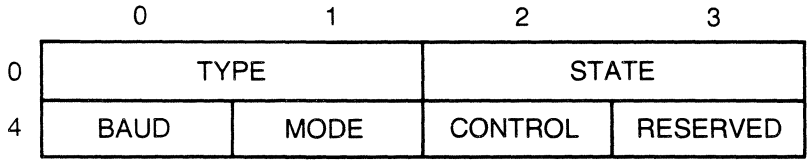


Figure 9-5. Port Driver GET/SET Table

Table 9-5. Port Driver GET/SET Table Fields

Field	Description
TYPE	Type of port, where: 0 = Undefined 1 = Standard serial driver 2 = Character I/O device 3 = Standard parallel driver
STATE	Bit map of port's state including error conditions. A bit set to 1 indicates the following conditions: Bit — Condition 0 — Ready to send a character (RTS) 1 — Character has been received 2 — Change in DSR or CD 3 — Parity error 4 — Overrun error 5 — Framing error 6 — Carrier present (CD) 7 — DSR
BAUD	Baud rate, as indicated by the following values: Value — Rate — Value — Rate 0 — 50 — 8 — 1800 1 — 75 — 9 — 2000 2 — 110 — 10 — 2400 3 — 134.5 — 11 — 3600 4 — 150 — 12 — 4800 5 — 300 — 13 — 7200 6 — 600 — 14 — 9600 7 — 1200 — 15 — 19200

Table 9-5. (Continued)

Field	Description
MODE	Bit map indicating word length, parity, and stop bits as follows:

Bits	Value	Mode
0-1	0	5 bits/word
	1	6 bits/word
	2	7 bits/word
	3	8 bits/word
2-3	0	0 stop bits
	1	1 stop bit
	2	1.5 stop bits
	3	2 stop bits
4-5	0	no parity
	1	odd parity
	3	even parity

CONTROL Bit map describing serial port control parameters. This field is intended for the use of the driver's SET function. A bit set to 1 indicates the following conditions:

Bit	Condition
0	Enable character transmission
1	Force DTR low
2	Enable character reception
3	Force break signal
4	Reset error
5	Force RTS low

9.2.6 SET--Change unit-specific information

Parameter: Address of SET parameter block.

Return Code:

E_SUCCESS	Write completed - no error
E_xxx	Driver specific error code

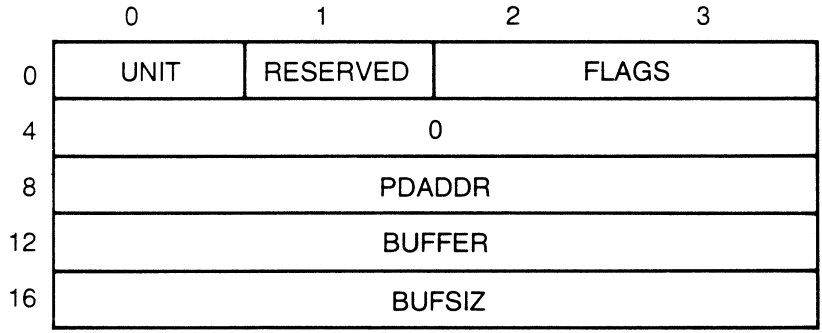


Figure 9-6. Port Driver SET Parameter Block

Table 9-6. Port Driver SET Parameter Block Fields

Field	Description
UNIT	Port driver unit number
FLAGS	Bit map of flags: Bits 0-14: Reserved Bit 15: 0 = System Address 1 = User Address
PDADDR	Process descriptor address of the process that initiated the SET call. This parameter is used to call the MAPU driver service if the buffer address is a User Address and the asynchronous portion of the driver is required to access the buffer.
BUFFER	Address of buffer containing information to be written to the driver's GET/SET Table. See Figure 9-5.
BUFSIZ	Size of buffer. This field determines the amount of information returned from the driver's GET/SET Table.

The Miscellaneous Resource Manager calls the SET entry point to set or modify unit-specific information in the port driver's GET/SET Table. The buffer indicated in the SET parameter block contains the information to be set. A buffer size parameter is passed by the calling process to determine the amount of information to be written to the driver's GET/SET Table.

See the explanation of the port driver's GET function for the description of the GET/SET Table.

End of Section 9

Printer Drivers

This section describes the driver interface for printer drivers. Printer drivers fall under the category of special drivers and are managed by the Miscellaneous Resource Manager.

10.1 Support for Printers

FlexOS supports multiple parallel and serial printers. Printers can be interrupt-driven or polled. The printer driver shipped with FlexOS is for a non-interrupt-driven parallel printer. You can implement a serial printer driver by installing a new unit to a serial driver or defining a new name (such as PRN: or LST:) for an existing serial unit.

A single printer driver can control multiple units of the same type of printer. The example driver supports up to four printers. FlexOS does not have a theoretical limit to the number of printing devices connected to a system.

The example printer driver uses the POLLEVENT driver service to emulate interrupts and maximize operating speed in a multitasking environment. POLLEVENT is described in Section 5.3, "Device Polling."

To allow multiple print jobs, the printer driver should be I/O reentrant at the driver and Resource Manager levels, and synchronized at the unit level. This means that bit 1 in the flags field of the printer's Driver Header should be set. See Section 4.2, "Driver Header" for a definition of the Driver Header. See Section 11.1 for a discussion of how the Miscellaneous Resource Manager protects drivers from user processes.

10.2 Printer Driver I/O Functions

This section describes the printer I/O functions accessed by the Miscellaneous Resource Manager through entry points in the printer driver's Driver Header.

The printer driver contains the SELECT, FLUSH, WRITE, GET, and SET functions. The READ function is meaningless for printers; it should return E_IMPLEMENT. The SPECIAL function is also not required by the printer driver and should return E_IMPLEMENT unless you support this SPECIAL functions in your driver.

See Section 4.4 for a description of the INIT, SUBDRIVE, and UNINIT driver installation functions.

10.2.1 SELECT--Enable the specified unit

Parameter: Address of SELECT parameter block

Return Code:

E_SUCCESS	Printer is enabled
IO_ERROR	Printer not enabled

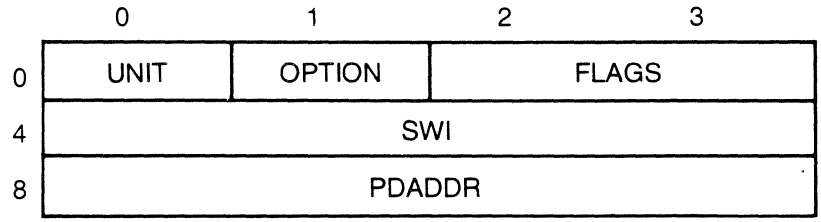


Figure 10-1. Printer Driver SELECT Parameter Block

Table 10-1. Printer Driver SELECT Parameter Block Fields

Field	Description
UNIT	Unit number of printer being enabled
OPTION	User-defined option
FLAGS	User flags field
SWI	Address of optional Software Interrupt Routine; this value is 0 if there is no SWI.
PDADDR	Process descriptor address of process attempting to open this device. This is not necessarily the process calling this entry point and therefore not the PDADDR used with the FLAGEVENT and FLAGSET driver services. Find the PDADDR of the process calling FLAGEVENT in the RLR field of the Driver Header. If an address is specified and it is a User Address, this is the PDADDR that must be used with the MAPU driver service.

The Miscellaneous RM calls SELECT to enable a specific printer unit for printing. A printer is selected before writing to it (if not previously selected), or if another unit was selected since last writing to this printer.

10.2.2 FLUSH--Disable Printer

Parameter: Address of FLUSH parameter block

Return Code:

E_SUCCESS	Printer is deselected
IO_ERROR	Printer not deselected

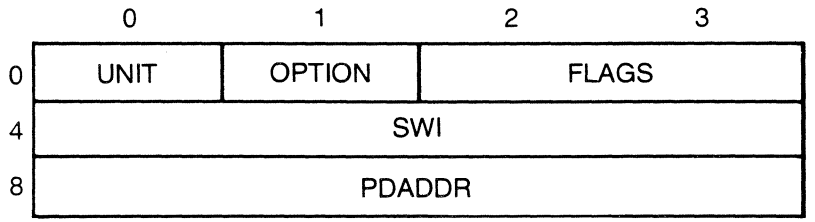


Figure 10-2. Printer Driver FLUSH Parameter Block

Table 10-2. Printer Driver in FLUSH Parameter Block Fields

Field	Description
UNIT	Unit number of printer to be disabled
OPTION	User-defined option
FLAGS	User flags field
SWI	Address of optional Software Interrupt Routine; this value is 0 if there is no SWI.
PDADDR	Process descriptor address of process attempting to write to this device. This is not necessarily the process calling this entry point and therefore not the PDADDR used with the FLAGEVENT and FLAGSET driver services. Find the PDADDR of the process calling FLAGEVENT in the RLR field of the Driver Header. If a User Address is specified, this is the PDADDR that must be used with the MAPU driver service.

The Miscellaneous Resource Manager calls FLUSH before writing to another unit connected to the same driver or before uninstalling the driver. If you are using a port driver to perform the actual printer I/O, the printer driver must call the FLUSH function in the sub-driver (port driver) to place it in a quiescent state.

Because writes are not allowed past the point of invoking FLUSH without first calling SELECT, any buffers not yet sent to the printer should be sent before FLUSH actually disables the unit.

10.2.3 WRITE--Write data to printer

Parameter: Address of WRITE parameter block

Return Code:

emask	Event mask for calling process to wait for printer to be ready for more characters
IO_ERROR	Unable to write to printer

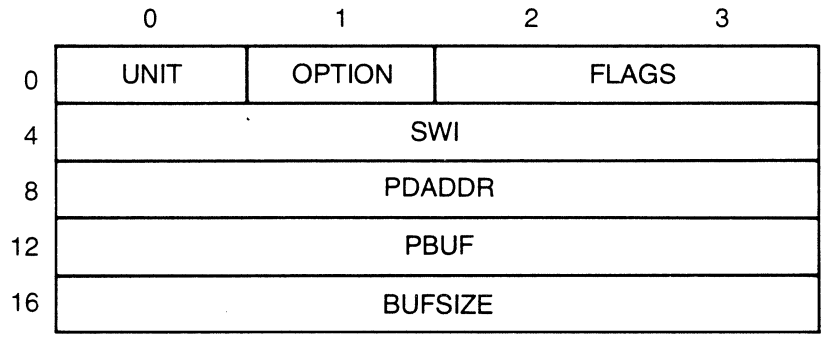


Figure 10-3. Printer Driver WRITE Parameter Block

Table 10-3. Printer Driver WRITE Parameter Block Fields

Field	Description
UNIT	Unit number of printer being written to
OPTION	User-defined option
FLAGS	User flags field
SWI	Address of optional Software Interrupt Routine; this value is 0 if there is no SWI.
PDADDR	Process descriptor address of process attempting to write to this device. This is not necessarily the process calling this entry point and therefore not the PDADDR used with the FLAGEVENT and FLAGSET driver services. Find the PDADDR of the process calling FLAGEVENT in the RLR field of the Driver Header. If an address is specified and it is a User Address, this is the PDADDR that must be used with the MAPU driver service.
PBUF	Pointer to buffer that holds characters to be written
BUFSIZE	Size of buffer indicated in PBUF

The Miscellaneous RM calls WRITE to output characters to a selected printer. As implemented in the FlexOS example driver, WRITE uses the POLLEVENT driver service to wait until the printer is ready, then outputs the character. The printer driver checks the status of the selected unit before calling POLLEVENT.

WRITE should first call FLAGCLR to make certain that the flag the driver obtained during its initialization is clear. It should pass the flag and the address of any SWIs to the FLAGEVENT driver service. FLAGEVENT returns an event mask that WRITE eventually passes back to the calling process on completion of the event.

WRITE latches a character to the output port for transmission to the printer. WRITE then calls POLLEVENT for an event mask with which to WAIT for the printer to become ready. POLLEVENT requires the address of status routine as a parameter. The status routine should return a non-zero WORD value if the unit is ready to receive a character; zero indicates that the unit is not ready. Your status routine should contain any delays required for carriage returns, form feeds, and any other device-related operations.

After all the characters have been written, WRITE must call FLAGSET and then return to the calling process the event mask obtained from FLAGVENT. Even though the event has already been completed (as signaled by FLAGSET), the call cannot be synchronous because POLLEVENT permits other tasks to run while the printer is busy.

See Section 5.1 for a description of the FlexOS flag system driver services. POLLEVENT is described in Section 5.3.

For an interrupt-driven printer, use the SETVEC driver service to establish an Interrupt Service Routine (ISR). Guidelines for using ISRs are presented in Section 5.7.

10.2.4 GET--Provide unit-specific information

Parameter: Address of GET parameter block

Return Code:

E_SUCCESS	Write to buffer completed - no error
E_xxx	Driver specific error code

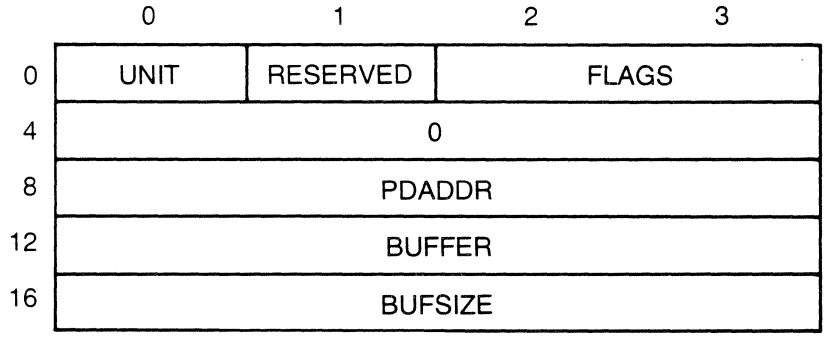


Figure 10-4. Printer Driver GET Parameter Block

Table 10-4. Printer Driver GET Parameter Block Fields

Field	Description
UNIT	Printer driver unit number
FLAGS	Bit map of flags: Bits 0-7: Can be defined by driver Bits 8-14: Reserved Bit 15: 1 = User Address 0 = System Address
PDADDR	Process descriptor address of the process that initiated the GET request. If the buffer address is a User Address and the asynchronous portion of the driver is required to access the buffer, this parameter is used to call the MAPU driver service.
BUFFER	Address of buffer in which to write information from the driver's GET/SET Table. See Figure 10-5, below.
BUFSIZ	Size of buffer. This field determines the amount of information returned from the driver's GET/SET Table.

The Miscellaneous Resource Manager calls the GET entry point to place information from the printer driver's GET/SET Table into the buffer at the address specified. The BUFSIZ parameter is passed to limit the amount of information to be obtained. If the buffer's size is less than the size of the table, only those fields that fit into the buffer are written there.

The GET and SET routines are not expected to return an event mask. The calling process should not return until the operation is complete. If the asynchronous portion of the driver is required to fulfill an I/O event in order to obtain the information, the GET and SET routines must perform their own WAIT and RETURN SVCs through the Supervisor interface described in Section 6.

Many of the GET/SET Table values cannot be determined until they are placed in the table by SET. GET/SET Table values are set by one process for use by another. For example, a parent process can configure a unit to print labels and then pass information about the width and length of the labels to its subprocesses through the appropriate table entries. The GET and SET SVCs can only be called if the printer driver unit is OPEN.

The format of the printer driver GET/SET Table is shown in Figure 10-5.

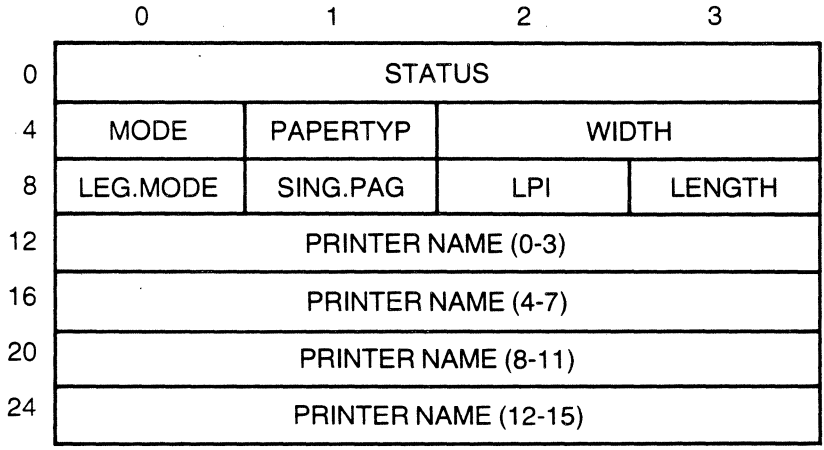


Figure 10-5. Printer Driver GET/SET Table

Table 10-5. Printer Driver GET/SET Table Fields

Field	Description
STATUS	Bit map of printer error codes--see Table 10-6
MODE	Current printer mode. This field specifies the printer typeface. This code may be replaced for other printer types, indicating the wheel-type on letter-quality printers, for example. The bit map for this field is the same for the LEG.MODE field (least significant bit is right-most): 0 boldface 1 graphics 2 italic 3 subscript 4 superscript 5 condensed 6 elongated 7 letter quality
PAPERTYP	This field indicates the type of paper currenly in use on the printer: 0 wide paper 1 letterhead 2 labels
WIDTH	Paper width, in columns, or dots if in graphics mode
LENGTH	Paper length, in lines
LEG.MODE	Printing modes supported by this printer (see MODE)
SING.PAGE	Set to non-zero if using a single-page-feed mechanism
LPI	Number of lines-per-inch
PRINTER NAME	This 16-bit field contains the printer's brand and model in ASCII.

Table 10-6 lists the Printer Status flag bits (least significant bit rightmost) and their meanings when set.

Table 10-6. Printer Status Bit Map

Flag Bit	Meaning
0	Printer unit off line
1	Out of paper
2	Select error
3	Initialization error
4	Illegal mode requested
5	Framing error
6	Internal buffer full
7	Waiting for XON

10.2.5 SET--Change unit-specific information

Parameter: Address of SET parameter block

Return Code:

E_SUCCESS	Write completed - no error
E_xxx	Driver specific error code

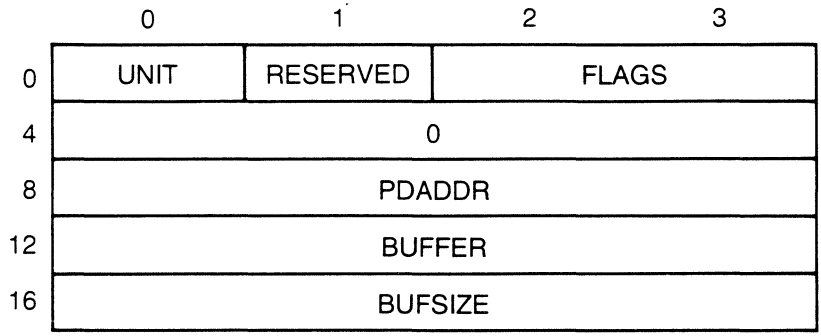


Figure 10-6. Printer Driver SET Parameter Block

Table 10-7. Printer Driver SET Parameter Block Fields

Field	Description
UNIT	Printer driver unit number
FLAGS	Bit map of flags Bits 0-14: Reserved Bit 15: 1 = User Address 0 = System Address
PDADDR	Process descriptor address of the process that initiated the SET call. This parameter is used to call the MAPU driver service if the buffer address is a User Address and the asynchronous portion of the driver is required to access the buffer.
BUFFER	Address of buffer containing information to be written to the driver's GET/SET Table. See Figure 10-5 above.
BUFSIZ	Size of buffer. This field determines the amount of information returned from the driver's GET/SET Table.

The Miscellaneous Resource Manager calls the SET entry point to set or modify unit-specific information in the printer driver's GET/SET Table. The buffer indicated in the SET parameter block contains the information to be set. A buffer size parameter is passed by the calling process to limit the amount of information to be written to the driver's GET/SET Table.

See the explanation of the printer driver's GET function for the description of the GET/SET Table.

End of Section 10

Special Drivers

This section describes the interface to special drivers. Special drivers interface to printers, plotters, ports and other devices not defined by FlexOS. Special drivers are managed by the Miscellaneous Resource Manager.

Special driver functions are available to application programs through standard Supervisor calls. The OPEN and CLOSE SVCs are mapped directly to the special driver's SELECT and FLUSH I/O functions. The READ, WRITE, GET, SET, and SPECIAL SVCs map directly to the corresponding entry points in the special driver's Driver Header.

In most cases, the Supervisor copies the user's parameter block into the System Area. The Supervisor and Miscellaneous Resource Manager then modify this copy of the user's parameter block before the special driver units are called with the address of the parameter block.

11.1 Special Driver Access

The Miscellaneous Resource Manager protects special drivers from user processes according to the level of access specified in the access flags parameter of the INSTALL SVC. INSTALL is described in the FlexOS Programmer's Guide. INSTALL's access flags are defined in the following table:

Table 11-1. Driver Access Flags

Flag	Description
Bit 0:	1 = SET allowed 0 = SET not allowed If flag bit 0 is 1, users can get the SET privilege to modify the driver's GET/SET table. If this bit is 0, the resource manager returns an error to users requesting the SET privilege at OPEN (the driver's SELECT function is not called).
Bit 1:	Reserved (must be set to 0)
Bit 2:	1 = WRITE allowed 0 = WRITE not allowed If flag bit 2 is 1, users can get the WRITE privilege to the device. If bit 2 is 0, the resource manager returns an error to users requesting the WRITE privilege at OPEN (the driver's SELECT function is not called).
Bit 3:	1 = READ allowed 0 = READ not allowed If flag bit 3 is 1, user can get the READ privilege to the device. If bit 3 is 0, the resource manager returns an error to users requesting the READ privilege at OPEN (the driver's SELECT function is not called).

Table 11-1. (Continued)

Flag	Description
Bit 4:	1 = Shared access allowed 0 = Exclusive access only If flag bit 4 is 1, multiple processes may have the same unit of the special driver OPEN at the same time. If bit 4 is 0, only one process may have the unit OPEN at any particular time. To enforce exclusive access, and indicate that the driver is synchronized at the unit level, you should set the Unit Level Interface Flag in the Driver Header (Bit 1) to 1. See Section 4.2, "Driver Header."
Bit 5:	1 = Removable driver 0 = Permanent driver If flag bit 5 is 1, the INSTALL SVC is allowed to remove the driver unit. The Miscellaneous Resource Manager does not allow the unit to be removed if it is currently OPEN or if another driver is using the unit as a sub-driver.
Bit 6:	1 = DEVLOCKs allowed 0 = DEVLOCKs not allowed The DEVLOCK SVC allows a process to temporarily restrict access to a driver it has opened. The process can restrict access to itself or processes within the same process family. DEVLOCK also allows the process to prevent the driver from being locked by other processes. This option is effective only if flag bit 4 is 1 (shared access allowed).

Table 11-1. (Continued)

Flag	Description
Bit 7:	1 = Shared access only 0 = Exclusive access allowed If flag bit 7 is 1, the Miscellaneous Resource Manager does not allow an exclusive OPEN of this device.
Bits 8-12:	Reserved
Bit 13:	1 = Force case to media default 0 = Do not force case to media default
Bit 14:	Used in interpreting the driver load file name given in the INSTALL SVC.
Bit 15:	Reserved

The Miscellaneous Resource Manager also restricts special driver access according to the OPEN SVC flags specified by the current process. For example, if a process opens the driver for exclusive access (OPEN flag bit 4 = 0), the Miscellaneous Resource Manager does not allow any other process to open the driver unless the driver has been INSTALLed for shared access (INSTALL flag bit 7 set to 1).

When a unit of a special driver is installed as a sub-driver, its higher-level driver can access any of its entry points. The higher-level driver must assume the responsiblities of controlling access to the special driver unit, acting, in effect, as the sub-driver's Resource Manager.

A controlling driver has the option of accepting or not accepting a special sub-driver's INSTALL options. The special driver unit attached as a sub-driver responds at its discretion to calls from the controlling driver.

11.2 Special Driver I/O Functions

This section describes the special driver I/O functions available to the Miscellaneous Resource Manager through entry points in the special driver's Driver Header. See 4.4 for a description of the INIT, SUBDRIVE, and UNINIT driver installation functions.

The READ and WRITE entry points are responsible for initiating the appropriate I/O request and calling the FLAGEVENT driver service to return an event mask (emask) to the calling process. The asynchronous portion of the special driver must complete the request by moving the appropriate data to or from the specified buffer. The asynchronous portion of the driver must then call the FLAGSET driver service to satisfy the outstanding request and return a completion code. See Section 5.1, "Flag System."

Each special driver is responsible for a single, well-defined table of information, the GET/SET Table. The GET and SET entry points read and write information in the table. The format of the GET/SET Table is dependent upon the special driver type. Any process can call a special driver's GET function at any time (through the LOOKUP SVC) without opening the driver. A user process can only call a special driver's SET function when the driver unit is open.

11.2.1 SELECT--Open a special driver unit for I/O

Parameter: Address of SELECT Parameter Block

Return Code:

E_SUCCESS	Successful operation
E_xxxxxxx	Special driver-specific error code. The OPEN is denied and this error code is returned to the user.

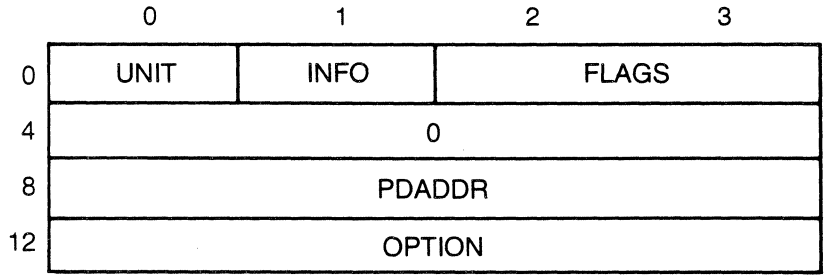


Figure 11-1. SELECT Parameter Block

Table 11-2. SELECT Parameter Block Fields

Field	Description
UNIT	Driver unit number
INFO	Unit information provided to the special driver by the Miscellaneous RM. 0 - This is the first OPEN 1 - This is a subsequent OPEN The Miscellaneous Resource Manager indicates if another process has the driver unit open (1) or if this is the first open (0) since either INSTALL or CLOSE (FLUSH).
FLAGS	OPEN call's flag field contents
PDADDR	Process descriptor address of process making OPEN call
OPTION	Contains the OPEN call's option field value in low 8 bits.

The SELECT function is called by the Miscellaneous RM to open the specified unit. The name field in the OPEN parameter block is translated by the Supervisor to identify which driver unit to open. The information passed to your SELECT function is the OPEN call's flags and option fields. These fields are passed unmodified. The flag's field options are listed in Table 11-3.

Table 11-3. SELECT Flags

Flag	Description
Bit 0:	1 = Delete file/set attributes 0 = No delete/set
Bit 1:	1 = Execute access 0 = No execute access
Bit 2:	1 = Write access 0 = No write access
Bit 3:	1 = Read access 0 = No read access
Bit 4:	1 = Shared access 0 = Exclusive access
Bit 5:	1 = Allow shared reads if shared 0 = Allow shared R/W if shared
Bit 6:	1 = Shared file pointer 0 = Unique file pointer
Bit 7:	1 = Reduced access accepted 0 = Return error on reduced access
Bits 8-12:	Reserved, must be 0
Bit 13:	1 = Force case to media default 0 = Do not affect name case
Bit 14:	1 = Literal name 0 = Prefix substitution allowed
Bit 15:	Reserved, must be 0

11.2.2 FLUSH--Close the specified special driver unit

Parameter: Address of FLUSH parameter block

Return Code:

E_SUCCESS	No hardware errors
E_xxxxxxx	Special driver specific error code

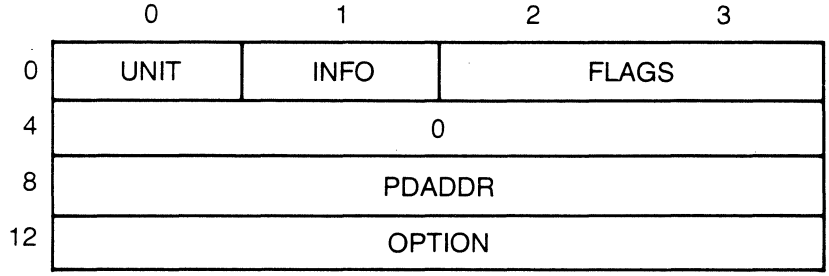


Figure 11-2. FLUSH Parameter Block

Table 11-4. FLUSH Parameter Block Fields

Field	Description
UNIT	Driver unit number
INFO	Close information provided by the Miscellaneous RM: 0 - This is the last CLOSE 1 - This is not the last CLOSE The Miscellaneous RM indicates if another process has the driver unit open (1) or if this is the last CLOSE. When last CLOSE is indicated, the driver unit is required to make itself quiescent.
FLAGS	Contents of CLOSE call's flag field: Bit 0: 1 = Partial close (flush only) 0 = Full close Bit 1: 1 = Do not close on error 0 = Close on error Bits 2-15 can be user parameters to the special driver's FLUSH function.
PDADDR	Process descriptor address of process making CLOSE call
OPTION	Contents of option field in the CLOSE parameter block

The Miscellaneous Resource Manager calls the FLUSH entry point when a process attempts to CLOSE a special driver unit it had previously opened. When the last CLOSE is indicated, the FLUSH routine must make the device quiescent. If your driver called has a sub-driver, you must call the sub-driver's FLUSH function.

The Miscellaneous Resource Manager passes the flags and option fields to FLUSH unmodified from the CLOSE SVC parameter block. When the partial close option is specified, the FLUSH function should only flush any buffers, but not actually close, the unit.

11.2.3 READ--Initiate request for data

Parameter: Address of READ Parameter Block

Return Code:

emask — Event mask as returned by FLAGEVENT. If an error occurs before the READ function can call FLAGEVENT to return the event mask, READ must call FLAGEVENT and then FLAGSET to return the error code. The error code must be returned to the calling process before READ returns to the Miscellaneous Resource Manager with an event mask.

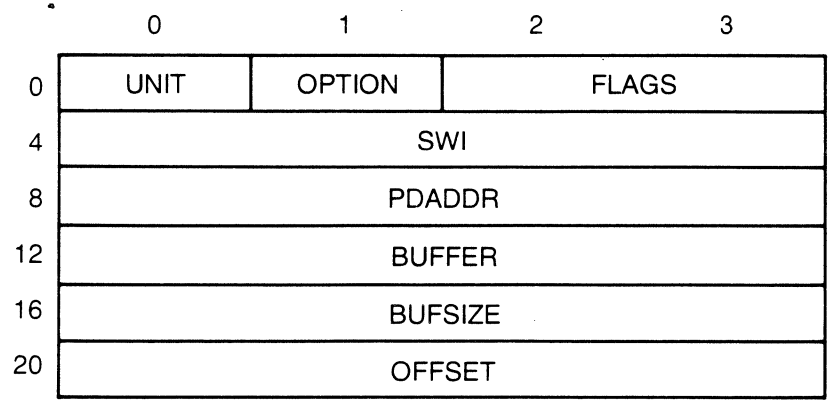


Figure 11-3. READ Parameter Block

Table 11-5. READ Parameter Block Fields

Field	Description
UNIT	Driver unit number
OPTION	Option field from the READ SVC
FLAGS	Flags field from the READ SVC. Bit 15 is turned on by the Miscellaneous Resource Manager if the buffer field is a User Address. All other flag bits are passed unchanged. Bit 15: 1 = Buffer is User Address 0 = Buffer is System Address
SWI	User-supplied software interrupt. The SWI is passed as a parameter to the FLAGEVENT driver service.
PDADDR	Process descriptor address of process that initiated the READ request. If the specified address is a User Address, this is the PDADDR that must be used for the MAPU function.
BUFFER	Data buffer address specified in READ call; the Supervisor checks the range before calling the READ function.
BUFSIZE	Size of buffer passed from READ SVC
OFFSET	Position in file relative to point indicated by value of bits 8 and 9 in READ SVC's flags field

The Miscellaneous Resource Manager calls the READ entry point when a process performs the READ SVC on a SELECTed special driver unit. The resource manager converts the READ SVC's file number into the READ parameter block's UNIT and PDADDR values. Most of the remaining parameter block contents are direct copies from the READ call's entries. The exception is flag bit 15, which the Miscellaneous RM sets to indicate whether the buffer provided is in User (1) or System (0) space. The buffer can be in System space when another driver calls the special driver unit with local buffers.

The offset field is used at the discretion of the special driver.

READ must initiate an I/O request, call the FLAGEVENT driver service, and return with an event mask. The special driver's asynchronous portion must complete the request by placing the appropriate data into the specified buffer. The driver then calls FLAGSET to clear the event from the system and return a completion code.

If an error occurs before READ can call FLAGEVENT, READ must call FLAGEVENT and then call the FLAGSET driver service to return the error code to the original calling process before returning to the Miscellaneous Resource Manager with the event mask. FLAGSET can be called from the synchronous portion of the driver's code.

11.2.4 WRITE--Initiate output of data

Parameter: Address of WRITE parameter block

Return Code:

emask — Event mask returned by the FLAGEVENT driver service. If an error occurs before FLAGEVENT is called, WRITE must first call FLAGEVENT and then call the FLAGSET driver service to return the error code before returning to the Miscellaneous Resource Manager with the event mask.

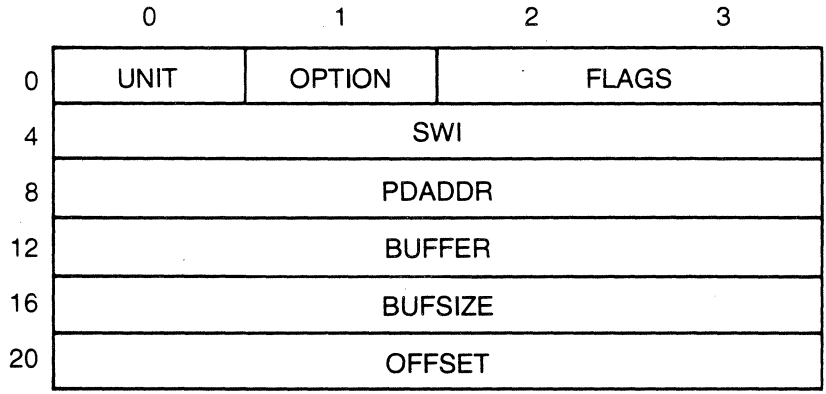


Figure 11-4. WRITE Parameter Block

Table 11-6. WRITE Parameter Block Fields

Field	Description
UNIT	Driver unit number
OPTION	User option field
FLAGS	Flags field as passed from WRITE SVC. The Miscellaneous Resource Manager turns on bit 15 if the buffer field is a User Address. Bit 15: 1 = Buffer is User Address 0 = Buffer is System Address
SWI	User-supplied software interrupt to be passed as a parameter to the FLAGEVENT driver service
PDADDR	Process descriptor address of process that initiated the WRITE request. If the specified address is a User Address, this is the PDADDR that must be used for the MAPU driver service.
BUFFER	Data buffer address specified in WRITE call; the Supervisor checks the range before calling the WRITE function.
BUFSIZE	Bufsiz field as passed from WRITE call; indicates number of bytes to write.
OFFSET	Position in file relative to point indicated by value of bits 8 and 9 in the flags field

The Miscellaneous Resource Manager calls the WRITE entry point when a process performs the WRITE SVC on a SELECTed special driver unit. The resource manager converts the WRITE SVC's file number into the WRITE parameter block's UNIT and PDADDR values. Most of the remaining parameter block contents are direct copies from the WRITE call's entries. The exception is flag bit 15, which the Miscellaneous RM sets to indicate whether the buffer provided is in User (1) or System (0) space. The buffer can be in System space when another driver calls the special driver unit with local buffers.

The offset field is used at the discretion of the special driver. WRITE must call the FLAGSET driver service upon completion to satisfy the outstanding event and return a completion code.

11.2.5 SPECIAL Entry Point

Parameter: Address of SPECIAL Parameter Block

Return Code:

emask Event mask returned by the FLAGEVENT driver service

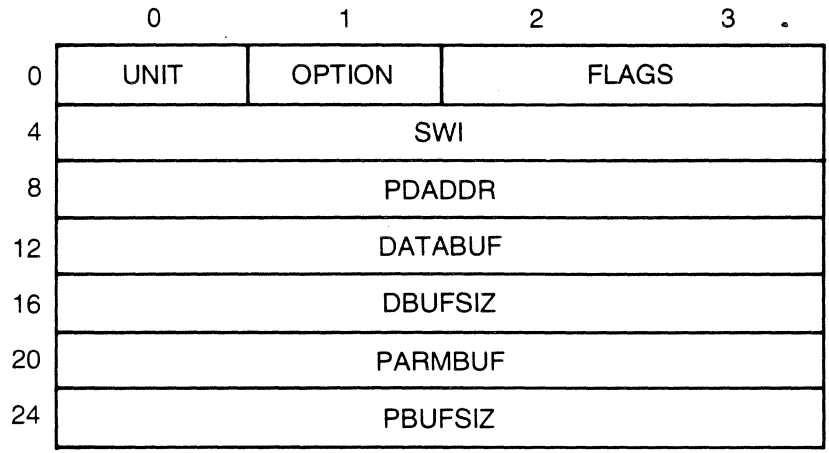


Figure 11-5. SPECIAL Parameter Block

Table 11-7. SPECIAL Parameter Block Fields

Field	Description
UNIT	Driver unit number
OPTION	Contents of the SPECIAL SVC's func field. The value of bits 7 and 6 indicate the data flow direction of the data and parameter buffers as follows: bit 7--parmbuf: 1 = write buffer, 0 = read buffer bit 6--databuf: 1 = write buffer, 0 = read buffer If no data or parameters are being provided, the corresponding bit is set to 0. The remainder of the bits indicate the SPECIAL function number.
FLAGS	Flags field as passed from the SPECIAL SVC. The Miscellaneous Resource Manager sets bit 15 if parameters came directly from User Memory. Bits 0-14: special driver-type specific Bit 15: 1 = User Address 0 = System Address
SWI	User-supplied software interrupt, passed as a parameter to the FLAGEVENT driver service.
PDADDR	Process descriptor address of process that initiated the SPECIAL request. If the specified address is a User Address, this is the PDADDR that must be used for the MAPU driver service.

Table 11-7. (Continued)

Field	Description
DATABUF	Value from SPECIAL call's databuf parameter. If DBUFSIZ is zero, this value is data; if DBUFSIZ is non-zero, this value is an address of a data buffer. The FlexOS Programmer's Guide instructs the programmer never to put an address in the data buffer.
DBUFSIZ	Size in bytes of data buffer; if zero, the DATABUF value is data rather than an address.
PARMBUF	Value from SPECIAL call's parmbuf parameter. If PBUFSIZ is zero, this value is data; if PBUFSIZ is non-zero, this value is an address of a data buffer. The FlexOS Programmer's Guide instructs the programmer never to put an address in the parameter buffer.
PBUFSIZ	Size in bytes of parameter buffer; if zero, the PARMBUF is data rather than an address.

The Miscellaneous Resource Manager calls the SPECIAL entry point when a user process calls the SPECIAL SVC on a previously SELECTed special driver unit.

If an error occurs before FLAGEVENT is called, the SPECIAL function must call FLAGEVENT and then call FLAGSET to return the error code to the original calling process before returning to the Miscellaneous Resource Manager with the event mask. FLAGSET can be called from the synchronous portion of the special driver.

See the description of the SPECIAL SVC in the FlexOS Programmer's Guide for rules on defining the SPECIAL driver function parameter block.

11.2.6 GET--Provide unit-specific information

Parameter: Address of GET parameter block

Return Code:

E_SUCCESS	Operation completed - no error
E_xxxxxxx	Driver type-specific error code

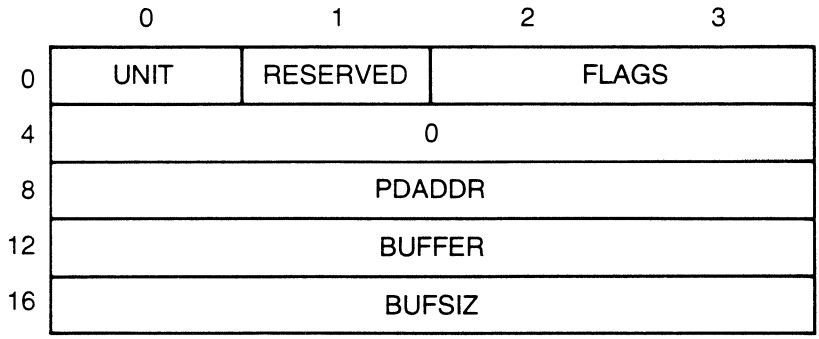


Figure 11-6. GET Parameter Block

Table 11-8. GET Parameter Block Fields

Field	Description
UNIT	Driver unit number
FLAGS	Bit map of flags Bits 0-7: Defined by driver Bits 8-14: Reserved Bit 15: 1 = User Address 0 = System Address
PDADDR	Process descriptor address of the process that initiated the GET request. If the buffer address is a User Address and the asynchronous portion of the driver is required to access the buffer, use this as the pdaddr value in your MAPU driver service call
BUFFER	Address of buffer
BUFSIZE	Size of buffer. This field determines the amount of information wanted.

You must define a single table structure for each type of special driver. The first field of this structure must be a 32-bit value that indicates the structure's size. When a user process calls the GET SVC, the special driver must return the requested information in a specified buffer in the defined table format. The GET and SET SVCs can be called only if the special driver unit is OPEN.

The Miscellaneous Resource Manager calls the GET entry point to place information from the special driver's GET/SET Table into the buffer at the address specified. The buffer size parameter is passed to indicate the amount of information requested. If the buffer size is less than the table size, fill in only those fields that fit completely.

The GET entry point is not expected to return an event mask. The calling process should not return until the operation is complete. If the asynchronous portion of the driver is required to fulfill an I/O event in order to obtain the information, the GET routine must perform its own WAIT and RETURN SVCs to complete the event. Drivers access SVCs through the Supervisor interface defined in Section 6, "Supervisor Interface."

11.2.7 SET--Change unit-specific information

Parameter: Address of SET Parameter Block

Return Code:

E_SUCCESS	Operation completed
E_xxx	Driver type-specific error code

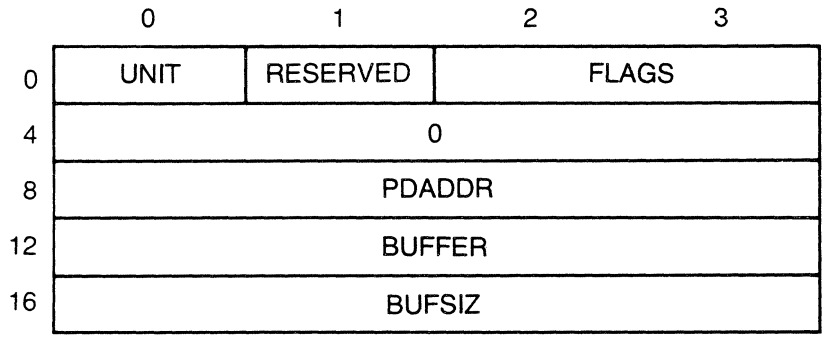


Figure 11-7. SET Parameter Block

Table 11-9. SET Parameter Block Fields

Field	Description
UNIT	Driver unit number
FLAGS	Bit map of flags Bits 0-7: Defined by driver Bits 8-14: Reserved Bit 15: 1 = User Address 0 = System Address
PDADDR	Process descriptor address of the process that initiated the SET SVC. If the buffer address is a User Address and the asynchronous portion of the driver is required to access the buffer, use this as the pdaddr value in your MAPU driver service call.
BUFFER	Address of buffer
BUFSIZ	Size of buffer; indicates the amount of information to set.

You must define a single table structure for each type of special driver. The first field of this structure must be a 32-bit value that indicates the structure's size. The GET and SET SVCs can be called only if the special driver unit is OPEN.

The Miscellaneous Resource Manager calls the SET entry point to modify information in the special driver's GET/SET table. The format of the table is specific to the type of special driver. The buffer indicated in the SET parameter block contains the information to be set. A buffer size parameter is passed to limit the amount of information being SET. If the size of the buffer is less than the size of the table, only those fields that fit within the buffer are SET.

SET is not expected to return an event mask. The calling process should not return until the operation is complete. If the asynchronous portion of the driver is required to fulfill an I/O event in order to change the state of the information, SET must perform its own WAIT and RETURN SVCs to complete the event.

End of Section 11

System Boot

This section outlines the steps required to cold boot a system. It describes boot and data disk formats and the FlexOS layout in memory. The section explains how to construct a loader and concludes with a description of the SYS utility, which transfers the loader to a disk's boot track.

Utilities used to generate a system that are specific to a given microprocessor are described in chip-specific supplements. These supplements are distributed with FlexOS.

12.1 Boot Overview

The boot procedure usually involves the following three steps:

1. A ROM reads the disk boot loader contained in the boot record, starting at track 0, sector 0 of the boot drive, then transfers control to it.

2. The disk boot loader reads the system image into memory starting at the first data cluster and continuing for n clusters, where n is the size of the operating system in clusters.

3. The disk boot loader then transfers control to the initialization routine in the operating system.

The boot loader reads the code into memory at the address specified by the code-load base address. The loader reads operating system data into memory at the address specified by the data-load base address. The loader then transfers control to the code-load base address, which should be the address of or a jump to the operating system initialization routines. Code-load and data-load base addresses are defined in Table 12-1, below.

The operating system initialization routines must perform any hardware and software initialization required by the operating system.

12.1.1 Data Disk Layout

The FlexOS disk layout is identical to the PC DOS disk layout, illustrated in Figure 12-1, below. The boot record, FAT, and root directory are all of variable size.

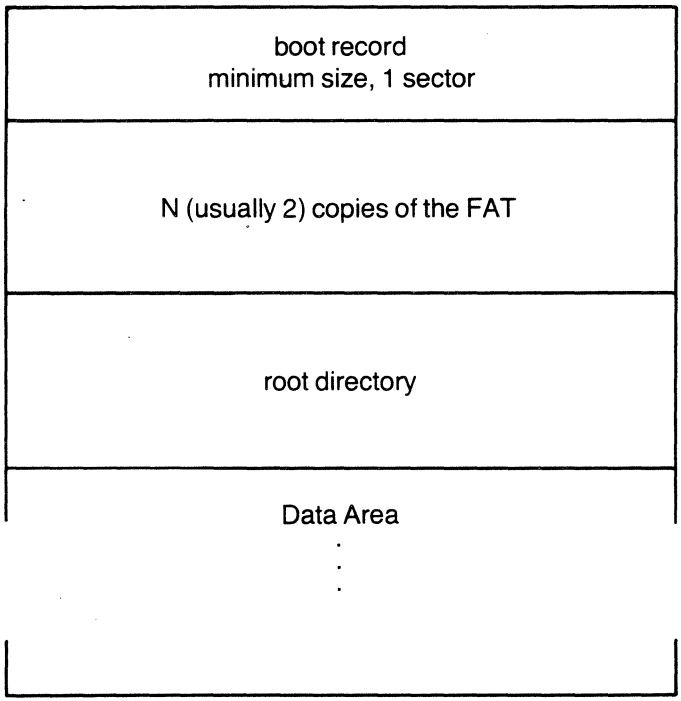


Figure 12-1. FlexOS Disk Layout

12.1.2 Boot Disk Layout

The FlexOS boot disk is a data disk which contains the operating system as illustrated above. The system file must be recorded under the file name FLEXOS.SYS, beginning at the disk's first data cluster and continuing for as many consecutive clusters as are required to store the complete operating system. Record the FLEXOS.SYS file with the System, Hidden, and Read-Only attributes.

12.2 Boot Record Format

The boot record (see Figure 12-2 below) contains the code needed to load FlexOS from disk into memory. It is a minimum of one physical sector in length. The boot record also contains information about where to load the various parts of the operating system and the sizes of each part. Table 12-1 defines the fields in the boot record.

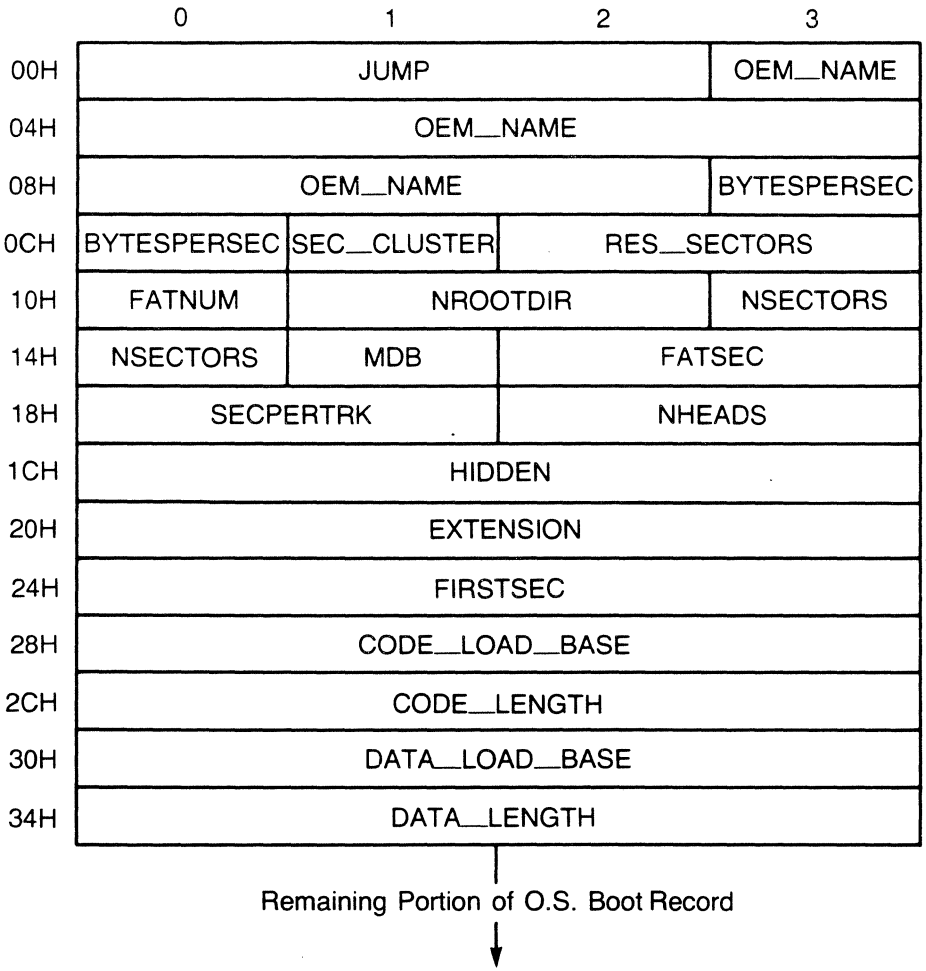


Figure 12-2. Boot Record

Table 12-1. Boot Record Fields

Field	Description
JUMP	A jump instruction to transfer control to an operating system's loader. See your chip-specific supplement for the description of the jump instruction.
OEM_NAME	The OEM name and version number identifying the boot record's operating system
BYTESPERSEC	Number of bytes per sector
SEC_CLUSTER	Number of sectors per file allocation unit (cluster) in a partition. This value must be a power of two.
RES_SECTOR	Number of sectors reserved by the operating system, starting at logical sector 0
FATNUM	Number of FATs in a partition
NROOTDIR	Maximum number of root directory entries in a partition
NSECTORS	Total number of sectors in a partition, including boot sector, directories, and reserved sectors. If this field contains zero, the EXTENSION field (see below) contains the total number of sectors.
MDB	Media Descriptor Byte. Describes the disk medium in terms of number of sides, number of sectors per track, and whether the medium is fixed or removable. Possible values for the MDB are defined in Table 8-5.
FATSEC	Number of sectors occupied by one FAT
SECPERTRK	Number of sectors per track in a partition
NHEADS	Number of heads in partition

Table 12-1. (Continued)

Field	Description
HIDDEN	Total number of sectors preceding a partition, including sectors occupied by the Master Boot Record.
EXTENSION	If NSECTORS contains zero, EXTENSION contains the total number of sectors in a partition. The EXTENSION field is used for partitions whose number of sectors is greater than can be stored in the one-word NSECTORS field.
FIRSTSEC	First sector of data area
CODE_LOAD_BASE	Address at which operating system code is to be loaded
CODE_LENGTH	Length, in bytes, of code segment
DATA_LOAD_BASE	Address at which operating system data is to be loaded
DATA_LENGTH	Length, in bytes, of data segment

The FORMAT utility fills in the fields from BYTESPERSEC through FIRSTSEC. The SYS utility (see Section 12.5) fills in the code and data load addresses and segment lengths.

12.3 Boot Loader Outline

Take the following steps in constructing a boot loader. The field names referenced below are defined in Table 12-1 above.

1. Calculate the number of physical sectors of code to read by dividing the value in the CODE_LENGTH field by the number of bytes in a physical sector. Add one physical sector if the division produces a remainder.

2. Read the operating system code into memory at the location specified by the CODE_LOAD_BASE field. The read always begins at the first sector of the first data cluster on the disk.

3. If the operating system code does not end on a physical sector boundary, the remainder of the sector is data. The data portion of the sector needs to be moved to the location in memory specified by the DATA_LOAD_BASE field.

4. Calculate the number of physical sectors of data to read by dividing the value in DATA_LENGTH, minus any data already read, by the number of bytes in a physical sector. Add one physical sector if the division produces a remainder.

5. Read the operating system data into memory at the location specified by the DATA_LOAD_BASE field plus the length of the data already read. The read begins at the next sector following the code reads.

If the operating system code and data are to be contiguous in memory, you can optimize the boot loader so that it performs one read that includes both the code and the data sectors.

12.4 The FlexOS Memory Image

Figure 12-3 shows the FlexOS memory image.

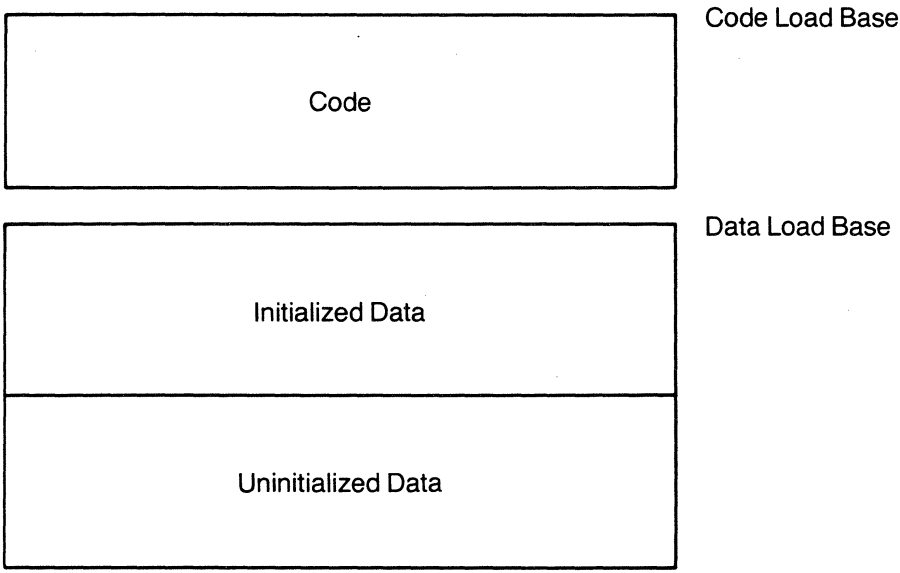


Figure 12-3. The FlexOS Memory Image

12.5 The SYS Utility

SYS transfers the operating system from the default to the specified drive or places the operating system onto the specified drive from a file. SYS modifies the destination drive's boot record to reflect the correct operating system Code Load Base, Code Length, Data Load Base, and Data Length fields.

SYS has the following syntax:

SYS d:

or

SYS d: d:filename.ext

The operating system image is placed in contiguous data clusters beginning at the first data cluster, Cluster 2. The first directory entry on the boot disk is FLEXOS.SYS. This file is recorded with the System, Hidden, and Read-Only attributes.

The header record on the specified input file is removed by SYS prior to placing the image on the disk. The information contained in the header record is used to update the proper fields in the disk's boot record.

End of Section 12

The FlexOS Standard Input and Output Character Sets

This appendix presents the characters sets supported by the FlexOS standard keyboard and standard console. The character sets are presented in the following order:

- 16-bit input characters (A.1)
- 8-bit input characters (A.2)
- 16-bit output characters (A.3)
- 8-bit output characters (A.4)

A.1 16-bit Input Character Set

This section defines the 16-bit character set supported by the FlexOS standard keyboard. The low order byte of a 16-bit input character is reserved for data. The high order byte can have the following values:

Table A-1. High-order Byte Values

Byte Value	Character Set
00H	ASCII character set. Includes DRI-standard US, Japanese, and European 8-bit character sets.
01H-7FH	Defined in Figure A-1, below.
80H-FCH	15-bit foreign language character sets, including KANJI.

When the high order byte of a 16-bit character is in the range from 01H to 7FH, the byte is defined as follows.

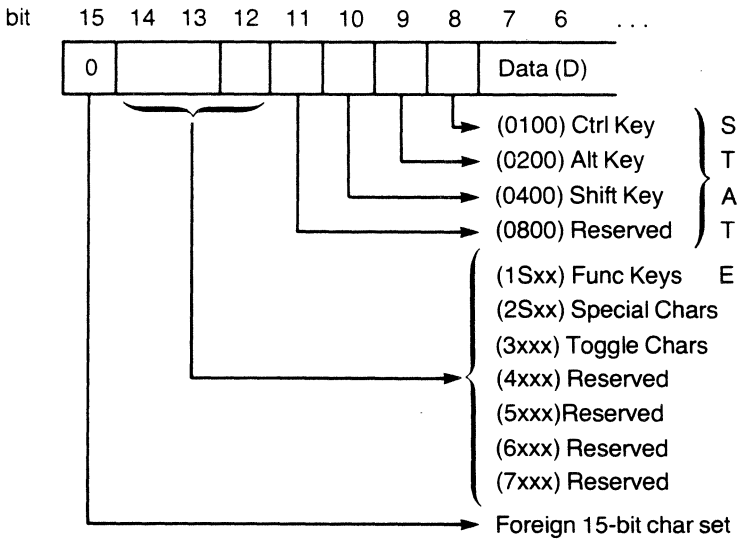


Figure A-1. High-order Byte Definitions for 01H to 7FH

Each defined bit field is described below. In the explanations, S refers to the state bits, bits 8-11.

- **STATE Bits:** The state bits indicate the status of the Ctrl, Alt, and Shift keys. When a state key is pressed along with another key, set the corresponding state bit or bits. If the ASCII standard specifies a code for the state and character key combination, the standard ASCII code should be generated without the state information. Examples of state bit use are:

  CTRL-C ==> 0003H
  SHIFT-p ==> P or 0050H
  CTRL-5 ==> 0135H
  CTRL-SHIFT-ALT-5 ==> 0735H

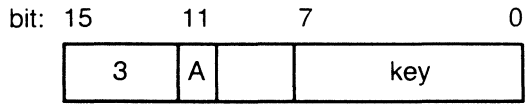
- **Func Keys:** Function key codes where you indicate the function key number in xx. Examples of function key codes are:

  FUNC 1 ==> 1001H
  CTRL-FUNC 1 ==> 1101H

- **Special Characters:** Special keys where the xx value should be generated for the key as follows:

Special Function		Cursor Movement		Numeric Keypad	
2S00	HELP	2S10	UP	2S30	ZERO
2S01	WINDOW	2S11	DOWN	2S31	ONE
2S02	NEXT	2S12	LEFT	2S32	TWO
2S03	PREVIOUS	2S13	RIGHT	2S33	THREE
2S04	PRINT SCREEN	2S14	PAGE UP	2S34	FOUR
2S05	BREAK	2S15	PAGE DOWN	2S35	FIVE
2S06	REDRAW (screen)	2S16	PAGE LEFT	2S36	SIX
2S07	BEGIN	2S17	PAGE RIGHT	2S37	SEVEN
2S08	END	2S18	HOME	2S38	EIGHT
2S09	INSERT	2S19	REVERSE TAB	2S39	NINE
2S0A	DELETE			2S3A	A
2S0B	SYSREQ			2S3B	B
				2S3C	C
				2S3D	D
				2S3E	E
				2S3F	F
				2S40	ENTER
				2S41	COMMA
				2S42	MINUS
				2S43	PERIOD
				2S44	PLUS
				2S45	DIVIDE
				2S46	MULTIPLY
				2S47	EQUAL

- **Toggle Characters:** Where your hardware supports toggle characters, generate the values for 3xxx according to the scheme:



where the fields are defined as follows:

A - action	0 - OFF 1 - ON
key	0x0 - Caps Lock 0x1 - Shift Lock 0x2 - Scroll Lock 0x3 - Num Lock 0x10 - Right Shift 0x11 - Left Shift 0x12 - Insert 0x13 - Control 0x14 - Alternate

Keys 0-3 should generate a character each time the user presses and releases the key.

Keys 10H-14H should generate a character when the user releases the key after pressing it along with another character

A.2 8-bit Input Character Set

The Console Resource Manager expects 16-bit keyboard input from the console driver. When the application specifies 8-bit character mode, the Console RM translates the 16-bit characters to 8-bit characters. The translation process may generate more characters than the console driver actually sends to the Console Resource Manager, as illustrated in Table A-2, below.

FlexOS supports the set of escape sequences only when the application is in 8-bit keyboard mode. Table A-4 lists the escape sequences supported.

Table A-2. Results of 16- to 8-bit Translation

16-bit Code	Result	Characters
00xxH	xxH	1
0100H-7FFFH	Converts to escape sequence except as follows (S=STATE bit values):	n
2S30	0x30H 0	1
2S31	0x31H 1	1
2S32	0x32H 2	1
2S33	0x33H 3	1
2S34	0x34H 4	1
2S35	0x35H 5	1
2S36	0x36H 6	1
2S37	0x37H 7	1
2S38	0x38H 8	1
2S39	0x39H 9	1
2S3A	0x41H A	1
2S3B	0x42H B	1
2S3C	0x43H C	1
2S3D	0x44H D	1
2S3E	0x45H E	1
2S3F	0x46H F	1
2S40	0x0DH RETURN or ENTER	1
2S41	0x2CH , (comma)	1
2S42	0x2DH - (minus)	1
2S43	0x2EH . (period)	1
2S44	0x2BH + (plus)	1
2S45	0x2FH / (divide)	1
2S46	0x2AH * (multiply)	1
2S47	0x3DH = (equal)	1
8000H-FCFCH	High byte, low byte	2

A.3 16-bit Output Character Set

The Console Resource Manager accepts 16-bit output characters through the WRITE SVC when in 16-bit screen mode. The 16-bit character codes provided are defined in Table A-3.

Table A-3. 16-bit Output Character Set

16-bit Value	Definition
00xxH	Same as 8-bit. Give these characters one character position on the screen. Characters in the range 80H-FFH are defined on a per-country basis.
8000H-FCFCH	16-bit language, such as KANJI. Give these characters two character positions on the screen. When modifying the FRAME, set the two character plane cells according to the key value and set the extension plane to indicate a two-cell character; that is, set bit 0 of both extension plane bytes to 1, set bit 1 of the first extention plane byte to 0, and set bit 1 of the second byte to 1.
01xxH-0FxxH	Alternate character sets. Implement these codes according to the following rules. Each character takes one character position. Typically, these characters are defined by the OEM extension field in a byte in a FRAME's extension plane. If an extension plane exists, the low byte is placed into the character plane while the low nibble of the high byte is placed into the low nibble of the extension plane.
1xxxH	Non-visible characters which take no space on the screen.
2xxxH	Editing functions. These functions are the equivalent of the escape sequences:

Table A-3. (Continued)

16-bit Value	Definition
2040	Enter Insert Character Mode
2041	Cursor Up
2042	Cursor Down
2043	Cursor Right
2044	Cursor Left
2045	Clear Display
2048	Cursor Home
2049	Reverse Index
204A	Erase to End of Page
204B	Erase to End of Line
204C	Insert Blank Line
204D	Delete Line
204E	Delete Character
204F	Exit Insert Character Mode
2064	Erase Beginning of Display
2065	Enable Cursor
2066	Disable Cursor
206A	Save Cursor Position
206B	Restore Cursor Position
206C	Erase Entire Line
206F	Erase Beginning of Line
2070	Enter Reverse Video Mode
2071	Exit Reverse Video Mode
2072	Enter Intensify Mode
2073	Enter Blink Mode
2074	Exit Blink Mode
2075	Exit Intensify Mode
2076	Wrap at End of Line
2077	Discard at End of Line

Table A-3. (Continued)

16-bit Value	Definition
3xxxH	Set cursor to xxx row (0 origin)
4xxxH	Set cursor to xxx column (0 origin)
50xxH	Set foreground color to color xx. Color codes are documented in the A.4, below.
51xxH	Set background color to color xx. Color codes are documented in the A.4, below.
52xxH-7xxxH	Non-visible characters. Take no space on screen.

A.4 8-bit Output Character Set

The Console Resource Manager converts the application's 8-bit output characters and escape sequences to 16-bit characters internally before calling the driver's WRITE function. The escape sequences are supported independently of the physical terminal type.

You can provide your own 8-to-16 bit conversion routine, accessible through the GET entry point in a console driver's Driver Header, to extend the character set. The extendability of the character set allows the implementation of SHIFT-JIS KANJI through the 8-bit character set in Japan. You might also modify the conversion routine to add escape sequences that switch to another character set.

You can support multiple country codes in the console driver. If you do, implement your selection routine in the driver's SET entry point using the value in the PCONSOLE table's COUNTRY field.

In the United States, FlexOS supports the IBM PC character set. In Japan, FlexOS uses the SHIFT-JIS character set. In Europe, the ISO standard ASCII character set is used.

While in 8-bit mode, the console's video attributes, such as cursor control, video blink, video intensity, and reverse video can be controlled through escape sequences sent through the WRITE SVC. The first character of an escape sequence is always the Escape character (ASCII character 27 or 1BH).

Table A-4, below, lists the escape sequences defined for FlexOS. This set of escape sequences is, with some exceptions, a superset of the set required by a VT-52 terminal. In the description below, <ESC> is followed by the function character. Blanks are used for clarity of presentation only.

Table A-4. FlexOS Escape Sequences for 8-bit Output

Function	Escape Sequence
Cursor Up	<ESC> A
Cursor Down	<ESC> B
Cursor Right	<ESC> C
Cursor Left	<ESC> D
Cursor Home	<ESC> H
Reverse Index	<ESC> I (upper case i)
Save Cursor Position	<ESC> j
Restore Cursor Position	<ESC> k
Set Cursor Position	<ESC> Y (r) (c) (r) = row + 32 (one character) (c) = col + 32 (one character) home = 0,0
Clear Display	<ESC> E
Erase to End of Page	<ESC> J
Erase to End of Line	<ESC> K
Erase Entire Line	<ESC> l (lower case L)
Erase Beginning of Display	<ESC> d
Erase Beginning of Line	<ESC> o
Insert Blank Line	<ESC> L
Delete Line	<ESC> M
Delete Character	<ESC> N

Table A-4. (Continued)

Function	Escape Sequence
Set Foreground Color	<ESC> b (c) where (c) = color (one character) 0 - Black 8 - Dark Gray 1 - Blue 9 - Light Blue 2 - Green 10 - Light Green 3 - Cyan 11 - Light Cyan 4 - Red 12 - Light Red 5 - Magenta 13 - Light Magenta 6 - Brown 14 - Yellow 7 - Light Gray 15 - White
Set Background Color	<ESC> c (c) where (c) = color (one character) 0 - Black 1 - Blue 2 - Green 3 - Cyan 4 - Red 5 - Magenta 6 - Brown 7 - Light Gray 8-15 are the same as 0-7 except the foreground blinks

Table A-4. (Continued)

Function	Escape Sequence
Enable Cursor	<ESC> e
Disable Cursor	<ESC> f
Enter Reverse Video Mode	<ESC> p
Exit Reverse Video Mode	<ESC> q
Enter Intensify Mode	<ESC> r
Exit Intensify Mode	<ESC> u
Enter Blink Mode	<ESC> s
Exit Blink Mode	<ESC> t
Enter Insert Mode	<ESC> @
Exit Insert Mode	<ESC> O
Wrap at End of Line	<ESC> v
Discard at End of Line	<ESC> w

End of Appendix A

Foreign Language Support

This appendix describes the FlexOS support for languages other than American English. FlexOS defines a console driver so an OEM can provide translation routines. In addition to this driver-level support, an OEM can translate or modify all messages displayed by FlexOS utilities.

Section B.1 describes provisions for foreign language support in a console driver. Section B.2 explains how to edit, recompile, and relink utility messages.

B.1 Console Driver Support

Support for foreign character sets exists in the console driver's SET and GET functions and in the extension plane of a FRAME. See Section 7, "Console Drivers," for a description of SET and GET and the FRAME data structure.

Through the SET function, a console driver can change the COUNTRY field in the PCONSOLE Table. The COUNTRY field contains a country code that determines which character set is being used. Applications can, through the GET and LOOKUP SVCs, obtain the country code from the PCONSOLE Table. Country codes are listed in Appendix C of the FlexOS Programmer's Guide.

Through the GET function, a console driver can provide the addresses of OEM-written character translation routines. The Console Resource Manager passes the address of the PCONSOLE Table to the GET function. The PCONSOLE Table has two 32-bit fields, CONVERT8 and CONVERT16, that can store pointers to translation routines. CONVERT8 can point to an 8-bit to 16-bit output translation routine. CONVERT16 can point to a 16-bit to 8-bit input translation routine. If these fields contain NULLPTR, the FlexOS standard conversion routines are called.

To support foreign language character sets, the console driver writer must implement an extension plane in his PFRAME. If the console driver supports virtual consoles, the extension plane must exist in the VFRAME also. FlexOS defines a byte in the extension plane to allow support for one-byte characters, two-byte characters, such as KANJI, or alternate character sets. See Section 7.2.1 for details.

B.2 Modifying Messages

Modifying the FlexOS messages consists of editing source message files, compiling those files, and linking the new object modules with a utility's code. The FlexOS utilities are written so code modules contain no messages and message modules contain only global symbols and messages.

The following message files are distributed with FlexOS:

- STDMSGS.C - contains all public messages, that is, all messages used by more than one utility. This file is provided as reference and is not used in the process of modifying messages.
- Set of files consisting of each message in STDMSGS.C in a separate file
- <utilityname>MSG.C - contains specific messages for a utility

Perform the following steps to create utilities that display modified messages.

1. Print STDMSGS.C, to use as a guide when editing individual message files.
2. Edit each of the public message files.
3. Edit the message file for each utility.

4. Run the batch file, CCMSGS.BAT. This file contains commands that do the following:

   - Compiles all public messages
   - For object files created by the Lattice C compiler, executes the COMB utility to change object files from Lattice format to a format usable by LINK-86 and LIB-86.
   - Runs LIB with the input file CCMSGS.INP using the I option. CCMSGS.INP contains a list of public symbols that LIB matches with corresponding message files to create a CCMSGS.L86 (CCMSGS.L68 for 68000-based systems), a library of standard object modules.

5. Run batch file, <utilityname>.BAT, for each utility. These files contain commands that do the following:

   - Compiles the <utility>MSG.C file containing a utility's messages
   - Runs LINK with a utility-specific input file, <utilityname>.INP. These input files contain a list of which code and message modules to link and a list of public symbols that LINK uses to extract appropriate messages from the CCMSGS file. LINK produces an executable file.

For 68000-based implementations that use SUBMIT rather than BATCH, SUBMIT files (file extension SUB) are provided.

The Window Manager and FORMAT utility distributed with FlexOS display text not contained in STDMSGS.C. Text strings for the Window Manager are stored in WMEXDATA.C (see Appendix C). Messages for FORMAT are stored in BOOT.A86 and HDBOOT.A86.

End of Appendix B

Modifying Windows

This appendix explains how you modify screen windows (virtual consoles) as set up by the FlexOS Window Manager.

You can modify the following window characteristics:

- size
- location on the screen
- attributes of windows, including borders
- fill characters
- number of windows to bring up at boot time
- startup command for each window
- text strings in window headers

A window can be as large as the limits of your physical console allow.

Distributed with FlexOS is the source code to the Window Manager, including two data files, WMEX.H and WMEXDATA.C.

WMEX.H contains the WNDWDESC structure, which describes a user window, and the WNDWSPEC structure, which describes a special window. A special window is a message or a status window. WMEXDATA.C contains data for both structures. WNDWDESC and WNDWSPEC are the only configurable window structures.

WNDWDESC describes a window's size, location, attributes, and fill characters. It also contains pointers to a text string for the window header and a pointer to the startup command line. WNDWSPEC contains pointers to text displayed in the message and status windows and stores the number of elements in a variety of different arrays of text strings.

In addition to configuration data, WMEXDATA.C contains the text strings pointed to in WNDWDESC and WNDWSPEC. You can translate, or otherwise modify these strings.

WMEX.H sets eight as the maximum number of user and special windows per physical console. Changing this number requires changing code in WMEX.H.

WMEXDATA.C defines eight windows: six user windows, a status window, and a message window.

In WMEXDATA.C, the variable WM0010, which stores the number of user windows at startup, is initialized to one. You can change this value to as many user windows as are defined in WMEX.H.

The variables WM0100, WM0110, and WM0120 in WMEXDATA.C define attributes and fill characters for the Desk Window. The Desk Window is the parent console for all user and special windows. Desk Window variables can also be modified.

In WNDWDESC, if you place a zero in the WD_RMAX (maximum number of rows) and WD_CMAX (maximum number of columns) fields, the Window Manager makes the window the size of the screen the Window Manager is running in. This is usually the size of the physical console.

In the WD_FLAGS field in WNDWDESC, you can change bits 0 (borders/no borders) and 1 (attributes/no attributes). Do not change bits 6 and 7.

End of Appendix C

Index

A

B

C

N

O

P