FlexOS™ 286

Release 1.42

Release Notes

1073-1001-004

Contents

Significant Differences

The following list summarizes the significant differences between FlexOS 1.42 and FlexOS 1.31:

- The installation process has been automated through the use of batch files.
- An new installation option is offered. The LOADFLEX.EXE command enables users to install FlexOS in a DOS partition, then start FlexOS from the DOS command line (see the Installation Instructions).
- The hard disk driver has been rewritten to improve performance. This new driver also allows partitions greater than 32Mb to be made bootable, and partitions other than the first one can be made bootable.

  Note: The new hard disk driver is renamed from ATHD.DRV to HD.DRV.
- The FDISK utility has been rewritten to support the new hard disk driver. See the Configuration Guide for information about FDISK.
- The floppy disk driver now implements **door open detection** to support AT-style drives (see Section 6).
- Large-model SRTL's (Shareable Run-time Libraries) are now supported and the documentation has been improved (see Section 11).
- There is a separate stand-alone debugger supplied on diskette **SBK 5**.
- An optional on-line, menu-driven help system is now available.

End of Section 1

Debug Disk

The diskette labelled **SBK 5** in the System Builder's Kit contains SASID286, a stand-alone debugger (a debugger that is executable and runs on a separate terminal) for debugging driver code. In order to use SASID286, you must generate a version of FlexOS you can run and debug from another terminal. The FLEX286.MAK file generates a working copy of FlexOS.

2.1 Using SASID386

Before you begin to use SASID, make sure that you have another serial terminal attached to the COM2 port on your development system. SASID286 uses this terminal to display information about activity taking place on the standard terminal.

Put diskette **SBK 5** in drive A and boot up the system. You will see the SASID286 prompt on the serial terminal screen. SASID286 is similar to the SID 286 debugger, described in the Programmer's Utilities Guide, and shares many commands with it. Table 2-1 lists SASID286 commands.

SASID286 also has built-in help. Simply press the question mark (?) to see a screen of help topics. Pressing two question marks gives you another screenful.

Table 2-1. SASID286 Commands

Display Memory	
b<address>,<length>,<address>	Compare memory
d<address>,<address>	Display bytes
dw<address>,<address>	Display words
dlw<address>,<offset>,<size>	Display linked list by words
l<address,address>	Disassemble code
sr<address>,<length>,<value>	Search for value
Examine Memory	
s<address>	Display and set bytes
sw<address>	Display and set words
f<address>,<address>	Fill memory bytes
fw<address>,<address>	Fill memory words
m<address>,<length>,<value>	Move memory block
Execute	
g<address>,<address>,<address>	Go at address until break
p<address>,<count>,<address>	Set passpoint
t<count>	Trace instructions
tw<count>	Trace without calls
u<count>	Trace without display
x	Display registers
c<address>,parm,parm...	Call a function

Table 2-1. (Continued)

Miscellaneous

h	Display symbols
h.symbol	Display symbol offset
h<value>	Hex-decimal conversion
h<value1>,<value2>	Hex arithmetic
n<name>,<address>	Add name to symbols
qi<port>	Input byte from port
qiw<port>	Input word from port
qo<port>	Output byte to port
qow<port>	Output word to port
y	Primitive stack dump

Note: A symbol or register can be used as an <address>.

End of Section 2

Shared Memory

3.1 Shared Memory Regions

FlexOS allows multiple processes to share common memory regions. Processes can also access specific physical memory locations, for dual ported RAM or system ROMs, through the shared memory driver services.

The processes can share data regions with drivers for fast communications in both protected and unprotected FlexOS environments, and multiple user processes can share data regions with each other. FlexOS grants access to shared memory only to those user processes with access rights established during system implementation.

There are two ways to access shared memory; through shared memory files, which work like pipes, and through the driver services SHMEM and UN_SHMEM.

Shared memory files are created with the CREATE SVC and have the "sm:" device name. A subsequent OPEN SVC provides and verifies access to the file. The GET SVC returns a valid address for the shared memory region, the CLOSE SVC disables access via this address, and DELETE releases the region. Each shared data file also contains a semaphore, so drivers and processes can synchronize usage through the READ and WRITE SVCs.

The Pipe Resource Manager disallows an open request of "sm:" devices by any process with an rnid <> 0. This prevents a process on a remote node of a network from gaining access to shared data. Note that pipes are different in this respect: processes on one node can access pipes on remote nodes.

Figure 4-1 shows the SHMEM table that can be examined with the LOOKUP SVC.

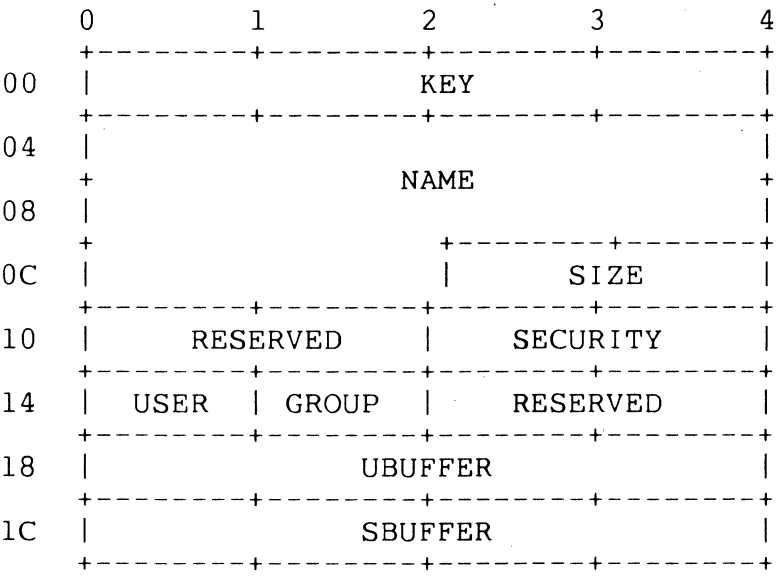


Figure 3-1. SHMEM Table

20H - Maximum Size of SHMEM Table

Table 3-1. SHMEM Table Fields

Field	Description
KEY	Unique ID
NAME	Name
RESERVED	Must be 0
SIZE	Size of memory area in bytes
SECURITY	Security word
USER	User ID of creator
GROUP	Group ID of creator
UBUFFER	User address of shared memory. This value is zero if the table was obtained through the LOOKUP SVC. You must use GET to obtain the address.
SBUFFER	System address of shared memory. This value is used by drivers and system processes independent of process context.
Device Type	0x11
Device Name	"sm:"
Table Number	0x11

3.2 Using Shared Memory Files

To create a shared memory region a user process performs the following calls:

```
fnum = s_create(0, flags, "sm:name", 0, security, size);
s_get(T_SHMEM, fnum, &shmem, sizeof(shmem))
buff_ptr = shmem.ubuffer;
```

BUFF_PTR now points to the shared memory.

If another user process wants to use the above shared memory file it performs the following calls:

```
fnum = s_open(flags, "sm:name");
s_get(T_SHMEM, fnum, &shmem, sizeof(shmem));
buff_ptr1 = shmem.ubuffer;
```

All references to *BUFF_PTR1 now access the named shared memory region.

A driver could give a user process access to a ROM of length LENGTH at address PHYS_ADDR by using the following calls:

```
struct
{
        LONG    link, pstart, plength;
} phys_mem = { 01, PHYS_ADDR, LENGTH };

sys_addr = (BYTE *)mapphys(&phys_mem, 1);
usr_addr = shmem(sys_addr, read_only_flag);
```

The user process would then use a SPECIAL() or GET() call to receive the user buffer address from the driver.

If two user processes need to synchronize access to a shared memory file they could each make the following calls:

```
s_read(0, fnum, "", 01, 01);     /* Get exclusive access  */
critical_code();                 /* Perform critical code */
s_write(0, fnum, "", 01, 01);    /* Release semaphore */
```

FNUM is the file number of the shared memory file obtained through the CREATE or OPEN calls.

When it no longer needs access to the shared memory file, the user process makes the call:

```
s_close(0, fnum);
```

FNUM is the file number that was attained by the CREATE or OPEN calls.

If the driver wants to remove user access to the shared memory it created it makes the call:

```
un_shmem(usr_addr);
```

usr_addr is the address obtained by the SHMEM() call.

3.3 Shared Memory Driver Services

A shared memory driver service must be used for a new process to gain access to shared memory. The driver obtains system memory by utilizing the MAPPHYS() or SALLOC() driver services, and then allows a user process access to memory through a SHMEM() driver service. The region is subsequently released with the UN_SHMEM driver service. This gives a user process direct control of memory related devices.

Note: OEM's must write a shared memory driver to support an application's use of shared memory, but FlexOS provides two subroutines that the driver can call.

3.3.1 SHare MEMory

The SHMEM driver service lets a user process address system memory while running in user space.

```
BYTE    *usr_addr, *sys_addr;
UWORD   flags;

usr_addr = shmem(sys_addr, flags);
```

Parameters:

flags	bit 0: 0 = Read/Write buffer 1 = Read Only buffer bits 1-15 are reserved
sys_addr	System address obtained through SALLOC() or MAPPHYS()

Return Code:

usr_addr	User buffer address. 0 indicates failure

3.3.2 UN_SHare MEMory

The UN_SHMEM() driver service reverses the function of a previous SHMEM() call. After this call, the user process gets an exception if it tries to access shared memory. If the user process passes an address to UN_SHMEM() that was not previously obtained through an SHMEM() call, it receives an error.

```
LONG     ret;
BYTE     *usr_addr;

ret = un_shmem(usr_addr);
```

Parameters:

usr_addr	User buffer address obtained through shmem()

Return Code:

ret0	indicated success; error code indicates bad usr_addr

End of Section 3

New Driver Service Calls

FlexOS 286 contains two new driver services that are not documented in the System Guide. These driver services allow FlexOS to more efficiently exploit memory management units (MMUs), which support paging of physical memory.

4.1 CSALLOC Driver Service

In previous versions of FlexOS, when calling SALLOC to allocate memory in the system address space, a driver could rely on the fact that this memory was always physically contiguous, so external devices under the driver's control such as DMA controllers which use physical addresses and bypass the MMU, would work properly.

This assumption is no longer true. If the memory to be allocated must be physically contiguous, CSALLOC must be used instead of SALLOC. A bit in the flags parameter determines whether contiguity is required.

The second reason for calling CSALLOC is to allocate memory which must be physically isolated from other system buffers. This use of CSALLOC is to exert control over a buffer which may be passed (by the DOS Application Environment) to a user process. If the allocation for the Application Environment were in the same page as the allocation for an important system data structure, the user process would have the potential to corrupt system data. The "isolate" bit in the CSALLOC flags word is used to control this area of memory protection. **Note:** In FlexOS 386, an isolated buffer starts on a 4Kb boundary and the allocation extends in multiples of 4Kb. The Intel^{R} 80286 processor does not support the same sort of hardware mapping, so a setting of the ISOLATE bit on a call to CSALLOC in FlexOS 286 is ignored.

C Interface for CSALLOC:

```
sysadr = csalloc (length, flags);
```

Parameters:

```
BYTE *      sysadr;         /* System address of memory block      */
                            /* allocated.  0 indicates no          */
                            /* memory available.                   */
ULONG       length;         /* Number of bytes to allocate         */
UWORD       flags;          /* Bit 0:
                                 1 = Physical contiguity required
                                 0 = Non-contiguity accepted
                               Bit 1:
                                 1 = Physical isolation required
                                 0 = Not required
                               Other bits must = 0.                */

/*'salloc(length)'is functionally equivalent to 'csalloc(length,0)'*/
```

4.2 CONTIG Driver Service

In previous versions of FlexOS, drivers that did DMA to and from physical memory addresses would call PADDR to convert the system address of the buffer to a physical address, with the driver assuming the whole buffer was physically contiguous starting from the address returned. In FlexOS 386, this assumption is no longer true. Therefore, CONTIG has been added to find the physical address of a buffer and the number of bytes that are physically contiguous from that point.

C Interface:

```
size = contig (buffer, length, &phyadr);
```

Parameters:

```
ULONG       size;       /* Number of bytes that are physically   */
                        /*   contiguous.                         */
BYTE *      buffer;     /* System address of buffer              */
ULONG       length;     /* Length (bytes) of buffer              */
BYTE *      phyadr;     /* Physical address of buffer (returned) */
```

The following conditions must be met before calling CONTIG for the first time for a given buffer:

1. The owner of the buffer must be mapped into memory (MAPU).
2. The buffer must have passed a MRANGE call.

After finding the number of bytes that are contiguous in the buffer, the driver can do DMA to or from that portion of the buffer. Note that the physical address of the buffer is returned via 'phyadr'. If the return 'size' is not equal to the 'length', then the buffer is not contiguous and more calls to CONTIG are required. For the next call, 'length' should be decreased by 'size', and the buffer address should be increased by 'size'. Then, assuming the process owning the buffer is still mapped in, another call to CONTIG can be made. Repeat this process until the returned 'size' and 'length' are the same.

End of Section 4

Removeable Subdrivers

FlexOS has the ability to remove subdrivers using the standard user DVRUNLK command and supervisor INSTALL function. A user can enter the subdriver device name in the DVRUNLK command to remove the subdriver from a driver. A programmer can use the INSTALL SVC with the option field set to 0 and the devname field set to the subdriver name address to remove a driver.

Subdrivers like drivers are set as removable or permanent in INSTALL flag bit 5. When the bit is set, the subdriver is marked as removable; otherwise, it should not be removable. The DEVICE Table's INSTAT field reflects the install status, whether permanent or removable.

For subdrivers, the fields are defined as follows:

0x00 - Not installed
0x01 - Requires subdriver
0x02 - Owned by Miscellaneous Resource Manager
0x03 - Owned by another driver
0x04 - Optional subdriver

Drivers are informed to remove a subdriver through the SUBDRIVE function entry point. This entry point is now used both to associate and disassociate a subdriver. To indicate which operation to perform, bit 10 in the Access field is set as follows:

Bit 10: 0 = Install subdriver
1 = Uninstall subdriver

The remainder of the Access flags remain as defined in Table 4-4, INSTALL Flags in the FlexOS System Guide.

The driver should then do whatever is necessary to remove the subdriver. Note, however, that the driver can ignore the request if the subdriver is currently in use. The following sample code illustrates a SUBDRIVE routine that handles both installation and removal of the subdriver.

```
LONG    s_subdrvr(pb)
DPB     *pb;
{
    PHYSBLK     *d;

    if(pb->dp_flags & 0x400){
        sfree(sdev[pb->dp_option]);
        sdev[pb->dp_option] = 0;
        return((((LONG)DVR_PORT << 16) | (LONG)DVR_SER));
        }

    ser_unit[pb->dp_option] = pb->dp_unitno;
    pt_hdr[pb->dp_unitno] = (DH *) pb->dp_swi;
    pt_unit[pb->dp_unitno] = pb->dp_option;

    d = sdev[pb->dp_option] =
        (PHYSBLK *) salloc( (LONG)sizeof(PHYSBLK));
    d->Qrear = d->Qfront = d->evpend =
        d->xoffed = d->Qlen = 0;

    return(E_SUCCESS);
}
```

The return code from the SUBDRIVE function should indicate the type of subdriver required or 0, if no subdriver is required.

Typically, the SUBDRIVE routine is not the only portion of the driver involved in the subdrive interface. For example, you should also free the resources such as flags, pipes and memory for data structures used to enable device I/O when the remove command is received.

As a general rule when removing subdrivers, everything done in INIT and SELECT to support device I/O should be undone in UNINIT and FLUSH, respectively.

End of Section 5

Open Door Detection

Section 8.1.2 of the System Guide describes the type of drives supported by the FlexOS Disk Resource Manager:

- non-removeable media (hard disk drive)
- removeable media with no door open support
- removable media with door open interrupt (DOI) support

Beginning with release 1.42, the Disk Resource Manager supports a fourth type: removable media with **door open detection** (DOD).

Drives that support removable media and do not support DOI incur high overhead since the Disk Resource Manager must periodically verify the media has not been changed. Door open detection provides support for removeable media without incurring high overhead. When the Disk Resource Manager decides to do a media check, it first checks the driver to determine if the door has been opened. If it has, the media is checked and then new media is logged in.

Since performance on removeable media only (RMO) drives would be unacceptably degraded if a verify was done on each entry into the caching code, the driver can specify how often the media should be checked (see Section 6.2, "Media Timed Verification").

6.1 Driver support for DOD

Beginning with release 1.42, the GET driver function call (see page 8-46 in the System Guide is modified to support door open detection. Figure 6-1 shows the modified parameter block.

GET--Provide unit-specific information

Parameter: Address of GET parameter block

Return Code:

E_SUCCESS	Successful operation
E_BADPB	Bad parameter block

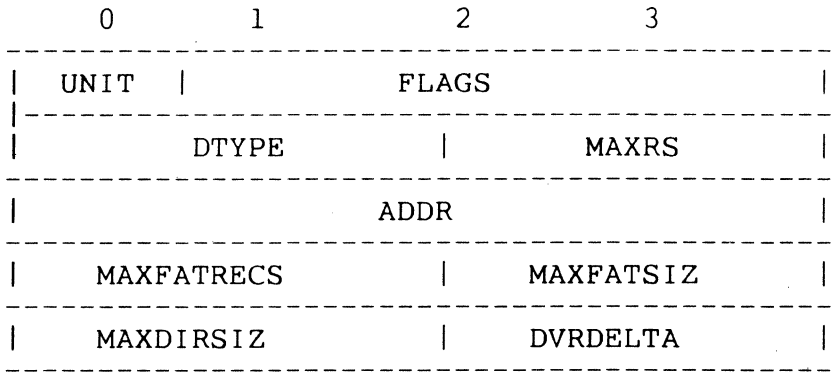


Figure 6-1. GET Parameter Block

Table 6-1. GET Parameter Block Fields

Field	Description
UNIT	Driver unit number
FLAGS	Reserved
DTYPE	Type of disk medium Bit 0: 1 = Removable media only 0 = Permanent media Bit 1: 1 = Open door support enabled 0 = Open door support diabled Bit 2: Reserved Bit 3: 1 = Door open detection enabled 0 = Door open detection disabled Bit 4: 1 = Media timed verification enabled 0 = Media timed verification disabled
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.

Table 6-1. (Continued)

Field	Description
MAXDIRSIZE	Maximum number of root directory entries.
DVRDELTA	Number of milliseconds to wait before performing media check if media hasn't been accessed.

Disk drivers that support door open detection should set the following bits in the **dtype** field:

bit 0 = on (Removable Media only)
bit 3 = on (Door open detection enabled)
bit 4 = on (Media timed verification enabled)

The driver must also support a new Special function as described below. The driver must check for an open door occurrence on all READ or WRITE calls and return E_READY if an open door is detected. After returning E_READY once, the driver must return it again only when the condition reoccurs.

Special Function 4 -- Get Open door information

Parameter: Address of SPECIAL parameter block

Return Code:

E_SUCCESS	Same media
E_UNITNO	Invalid unit number
E_BADPB	Bad parameter block
E_NEWMEDIA	New media
E_DOOROPEN	Door has been opened

Figure 6-2. SPECIAL Function 4 Parameter Block

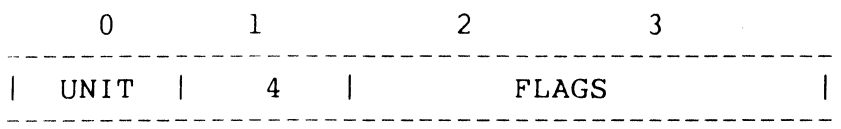


Table 6-2. SPECIAL Function 4 Parameter Block Fields

Field	Description
UNIT	Driver unit number
OPTION	Special function number (in hex)
FLAGS	Bits 0-15 are reserved

Special function 4 returns E_NEWMEDIA when the driver verifies there is new media in the drive. E_DOOROPEN is returned when there may be new media in the drive.

On a E_DOOROPEN error, the Disk Resource Manager performs a media check to verify if there is new media in the drive. If there is new media, or E_NEWMEDIA was returned, the Disk Resource Manager performs a relog operation on the unit. Once the driver returns

E_NEWMEDIA or E_DOOROPEN, it must not return either value until it determines there is new media in the drive, or the drive door has been opened again.

The SELECT function should only report E_NEWMEDIA or E_DOOROPEN if the condition arises while executing the select code. If either the E_NEWMEDIA or E_DOOROPEN error is returned from a SELECT call, the Disk Resource Manager attempts one more SELECT call before reporting E_MEDIACHANGE to the user.

6.2 Media Timed Verification (MTV)

The Disk Resource Manager supports **media timed verification** (MTV) to increase the performance of removable media. Media timed verification relies on the fact that it takes some minimum amount of time for a user to remove the media and replace it with new media. Using this knowledge, the Disk Resource Manager can verify the media every N seconds (provided N is less than the minimum) and not miss a media change. The media is not verified when it is not in use, but is checked only during caching or I/O operations on the given media.

Media timed verification is most useful on removeable media only (RMO) units where the Disk Resource Manager incurs considerable overhead verifying the media has not changed. Media timed verification can be useful on units that implement door open detection if executing the driver's DOD code takes excessive time. Media timed verification is of no use on units that use door open interrupts because there is no overhead for the Disk Resource Manager to check the door open flag.

6.3 Driver support for MTV

Driver support for media timed verification is provided through the Disk Driver GET parameter block (see page 8-45 in the System Guide). Beginning with release 1.42, this parameter block is modified as follows:

- an additional bit (bit 4) is defined in the **dtype** field. Bit 4 must be set to 1 (on) to enable media timed verification; otherwise it is disabled.
- the ULONG value **dvrdelta** is added to the end of the parameter block. The dvrdelta field contains the number of milliseconds the Disk Resource Manager waits before performing a media check.

When calling GET on a unit that supports media timed verification, the driver must set both bit 4 in the **dtype** field and the number of miliseconds for dvrdelta.

Note: The Disk Resource Manager does a media check only when the media has not been accessed in the time span given by dvrdelta, but otherwise performs no check.

End of Section 6

The SYSTAB Utility

FlexOS contains a new utility named SYSTAB that displays the current status of the following system tables described in Section 8 of the FlexOS Programmer's Guide:

- CONSOLE
- DEVICE
- ENVIRON
- MEMORY
- PIPE
- PROCDEF
- PROCESS
- SHMEM
- SYSDEF
- SYSTEM
- VCONSOLE

To use SYSTAB, enter the command:

```
C>SYSTAB
```

which displays the main menu of available commands and the command-line parameters that control the display operation.

End of Section 7

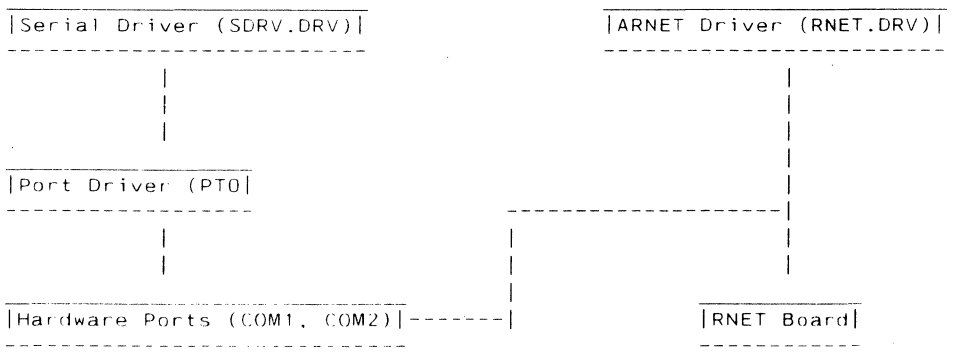
RNET Serial Driver

RNETHSD.DVR is a high-speed serial device driver provided in executable form with the Programmer's Toolkit, and in both executable and source form with the System Builder's Kit. The RNETHSD.DRV source code is fully commented, and illustrates the basic structuring of a device driver into synchronous and asynchronous portions.

RNETHSD.DVR is the driver for the ARNET Multiport™ Board, which is an RS-232 serial communications board designed for use with IBM AT and compatibles.

If you need high-speed serial communications, you can use RNETHSD.DRV as an replacement for the standard serial driver SDRV.DRV. To do this, you edit the source code, setting the conditional compilation switch, then recompile to make RNETHSD.DRV communicate directly with the hardware ports COM1 and COM2. In its default configuration, RNETHSD.DRV communicates with the ARNET board (see figure on the following page).

Note: Normally, the serial driver SDRV.DRV communicates with the port driver PT0:, which then communicates directly with the hardware ports. If you use RNET.DRV, it communicates directly with the hardware ports and overrides the port driver, thus disabling the capability of using a serial terminal or a mouse.



End of Section 8

Limits on Processes

The Intel 286 microprocessor architecture imposes on FlexOS certain limits on system values such as the number of files that can be opened, the maximum number of concurrently scheduled ASR's, and the size of internal memory pools. These values are all configurable by editing the system files CONFIG.C, CONFIG.H, and ACONFAT.A86.

If you encounter problems because these numbers are too small, simply edit the system configuration files and increase the values. For example, you can increase the size of the internal memory pools, or the number of entries for the File Number Table, then recompile and relink the system.

Also, adhering to the following guidelines may help you avoid problems related to the limitations imposed by these default values:

- Use the HSET command (see the Programmer's Utilities Guide, Appendix L) on all executable files to set the code group Descriptor in the .286 file header to 09 (shared code).
- Conserve space in the File Number Table by closing all unused files numbers, including the standard file numbers (stdin,stdout,stderr,stdcmd) associated with each process.

End of Section 9

Documentation Errata

This section contains corrections and/or additions to the indicated manuals. You should annotate your manuals accordingly.

FLEXOS™ 286 PROGRAMMER'S UTILITIES GUIDE, (1073-2043-001)

In Section 7, change all references to the .CMD filetype to .286 filetype.

On page 7-18 at the bottom, change the sentence

"LINK 86 makes multiple passes through the library index when attempting to resolve references from other modules."

to read as follows:

"LINK 86 makes multiple passes through the library index when attempting to resolve references from other modules within the library."

On page 10-5 in the explanation of the fourth example, change "12800 bytes" to "128K bytes".

FLEXOS™ SYSTEM GUIDE, First Edition: November 1986 (1073-2013-001)

In section 3.2 on page 3-3, change the reference in the fourth paragraph from FlexOS User's Guide to FlexOS Configuration Guide.

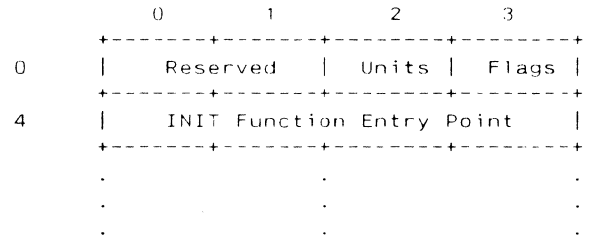
In Listing 4-1 on page 4-2, change the line:

```
UWORD  dh_reserved   ;/* Reserved                              /*
```

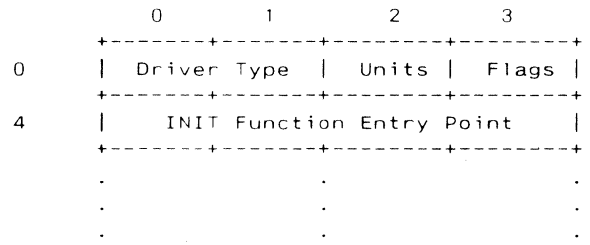
to read as follows:

```
UWORD  dh_dtype    ;/* Type of Driver                        /*
```

Also change the top line of Figure 4-2 on page 4-3 from:



to the following:



and add the following explanation to Table 4-1 on page 4-4:

Driver Type — A word-length description of the type of the driver as listed in Table 4-3.

Add the following to the explanation of the fields in Table 4-2, "Driver Header Synchronization Flags" at the bottom of page 4-6:

> Flag bit 4 controls delimited reads. Set bit 4 to zero to use the delimited read routine supplied by the Console Resource Manager. On systems that do not have a Disk Resource or Console Resource Manager, set bit 4 to one to have a user-supplied driver perform the delimited read.

On page 4-10, add the following entry to Table 4-3:

75 Mouse Driver

On page 4-11, make the following changes to Table 4-4:

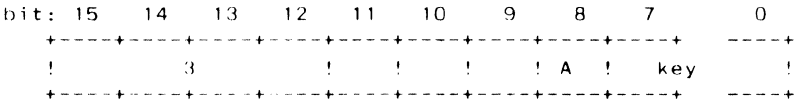
- Add bit 10 as follows:

  Bit 10: 0 = Install subdriver
  1 = Uninstall subdriver

- Change "Bits 10-12" to "Bits 11-12".

In Figure A-1 on page A-2, note that the **state** bits (8-11) do not apply to the Toggle characters.

The figure on page A-3 is incorrect. It should be identical to the figure on page A-11 of the Programmer's Guide:



FLEXOS™ USER GUIDE, Second Edition: August 1987 (1073 2004 002)

On page 1-1, change the paragraph:

"An **option** lets you modify what a command does. Multiple options are separated from each other by hyphens (-)."

to read as follows:

An **option** lets you modify what a command does. Multiple options are separated from each other by hyphens (-). The hyphen is the default **switchar**, or options separator (see Table 1-6). The switchar must be a single character; it cannot be any of the following:

/ (backslash) , (comma) ; (semicolon) = (equal sign)

On page 1-10 in Section 1.8.4, change the example to read as follows:

```
A>DEFINE PROMPT=Your wish is my command$g
```

On page 3-2, change the example at the top of the page to read as follows:

"If you are on drive B and you want to run CLEANUP.BAT from drive A, enter the command:

```
B>A:CLEANUP
```

CLEANUP.BAT runs its commands against the files on drive B. When it is finished, it leaves you in the root directory of drive B.

On page 3-3, make the following changes:

- In the middle of the page change:

```
DEL %1
DIR %1
DEL %2
DIR %2
```

  to read:

```
DIR %1
DEL %1
DIR %2
DEL %2
```

- At the bottom of the page change:

```
DEL %2
DIR %2
DEL %1
DIR %1
```

  to read:

```
DIR %2
DEL %2
DIR %1
DEL %1
```

On page 3-22, delete the sentence that reads:

"VERIFY does this whether it is on or off."

FLEXOS™ USER'S REFERENCE GUIDE, First Edition: August 1987 (1073-2064-001)

On page 1-20, change all references of SHELL.EXE to SHELL.286.

On page 1-26, change the example

```
A>COPY B:MYFILE
```

to read as follows:

```
A>COPY B:MYFILE.TXT
```

On page 1-43, make the following changes:

- In the first paragraph under **Examples** delete the sentence that reads:

  "FlexOS waits for you to enter another command".

- Change the example at the bottom of the page to read as follows:

```
ECHO OFF
TIME -D
ECHO Although ECHO is OFF, this message
ECHO is displayed.
REM  This line will not be displayed.
ECHO ON
REM  When ECHO is ON, remarks are displayed
REM  as well.
```

On page 1-51, delete the sentence at the bottom of the page:

"Items in a FOR command line must be filenames."

On page 1-53 and 1-54, disregard any references to **physically** formatting the hard disk. You should always assume that a hard disk has been physically formatted by the manufacturer. The documentation regarding **logically** formatting a hard disk is correct.

On page 1-67 in the explanation of the example, change references to "noname" to "No-Name".

On page 1-92 in the **Explanation**, change the sentence:

"If you do not specify a path for the source drive, the files are restored to the current directory."

to read as follows:

"If you do not specify a path for the source drive, files backed up from the current directory are restored to the current directory."

On page 1-101 make the following changes:

- Change the example command line:

  A>**SORT < PARTTIME.LST >> FULLTIME.LST >PRN**

  to read as follows:

  A>**SORT < PARTTIME.LST >> FULLTIME.LST**

- Delete the sentence:

  "This file is redirected to the printer device PRN:."

- On page 1-109, delete the sentence that reads:

  "VERIFY does this whether it is on or off."

End of Section 10

Change Pages

The pages following this one are change pages for the Programmer's Uitilies Guide, (1073-2043-001).

Replace all the pages in Appendix B and Appendix H.

Creating Shared Runtime Libraries

B.1 Shareable Runtime Libraries

This appendix describes the procedures for creating and modifying shareable runtime libraries (SRTLs). SRTLs allow multiple users to share a single copy of library code at runtime. This makes it unnecessary for each user to store library code in a command file. When libraries are shared, only references to the library code are linked with the user's object files.

Before attempting to create or modify shareable runtime libraries, you should be familiar with the 80286 architecture, memory models, and calling conventions. You should also be familiar with conventions used when writing reentrant code.

You can write most shareable runtime library code in C, or most other high level languages. The only code that cannot be written in a high level language is the **transfer vector** code, which handles the calls from the user program to the SRTL routines. Transfer vector code must be written in 80286 assembly language.

See Section 7 in this manual for a description on how to link shareable runtime libraries with your object files.

B.2 SRTL Components

A SRTL consists of two types of files:

- A shareable runtime library file which contains the basic shareable object code created by LIB-86. The filetype is .L86.
- An XSRTL code file which contains an executable version of the SRTL created by LINK 86. The filetype is .SRL.

This appendix describes how to create these two files.

B.3 Creating a SRTL

Creating a SRTL involves the following general steps:

1. If you plan to compile the SRTL using a Large memory model[1], you must modify the source of the library routines to use the proper calling conventions.
2. Create the transfer vectors.
3. Compile the source for the library routines and transfer vectors to create object modules.
4. Create the LIBATTR module.
5. Use LIB-86 to create the SRTL.
6. Use LINK 86 to create an executable SRTL (XSRTL).

These steps are described in detail in the following sections.

B.3.1 Modifying the Source

To create a C language SRTL (in Large memory model) with the recommended transfer vectors, you must modify the SRTL library source code. Two modifications are required:

1. The formal parameter lists of the entry point routines in the SRTL must include three extra words of parameters. These parameters provide "placeholders" for the extra information inserted onto the stack by the transfer vector (see Section B.3.2).
2. Any intra-library calls to the entry point routines must have three words added to the actual parameter list, preceding the "real" parameters, so local calls emulate the stack activity of external calls through the transfer vectors. External calls must call the transfer vector symbol name, but are otherwise unchanged.

---

[1]The type of memory model you use is dependent on your compiler. As used here, Large refers to a memory model that allocates multiple code, data, and heap segments, as well as a separate segment for the stack.

If transfer vectors are used, intermediate names must be put in the library. This is also true for Small memory model[2] SRTLs using a transfer vector at the beginning of the library in order to make all entry points constant.

Using C conditional compilation statements, a single set of sources could be used for both shareable and nonshareable libraries, as shown below:

```
#ifdef NonShareable
        strcpy (to, from)
                char *to, *from;
#else
        strcpy (d1, d2, d3, to, from)
                WORD d1, d2, d3;
                char *to, *from;
#endif
   .
   .
   .
#ifdef NonShareable
        strcpy (source, target);
#else
        strcpy (0, 0, 0, source, target);
#endif
```

[2]As used here, Small refers to a memory model that allocates a single segment for the program's data, heap, and stack.

B.3.2 Creating the Transfer Vectors

There are many possible transfer vector calling conventions for handling calls between user programs and SRTLs. The calling conventions you use depend on your application and your programming style. However, the transfer vector calling conventions described here are the only ones tested and supported by Digital Research.

There are two types of transfer vectors: User and SRTL.

User Transfer Vector

If the user program uses a Large memory model SRTL, calls to the SRTL routines must first pass through a transfer vector. This transfer vector, referred to as a User Transfer Vector, stores the value of the user program's DS register and loads the SRTL's DS register prior to entering the SRTL. Upon exiting the SRTL, the transfer vector restores the user program's DS register. The user transfer vector must reside in the user program's code space and must have a separate entry for each entry point into the SRTL.

See Section B.5 for more information on User Transfer Vectors.

SRTL Transfer Vector

If either memory model is being used, you can create an optional SRTL Transfer Vector that resides in the SRTL. This transfer vector is a collection of jumps that establish the SRTL entry points at fixed locations. By making an entry point into the SRTL a fixed location (constant virtual address), user programs do not have to be relinked each time a change is made to the library.

There must be an entry in the SRTL Transfer Vector for each entry point into the SRTL.

Note: To make SRTL entry points constant, any object modules containing the SRTL transfer vector must appear immediately after the LIBATTR module in the SRTL.

B.3.3 Creating the Object Modules

This step involves compiling the library source files to create library object files. Be sure to set any compiler options required to generate the object files using the correct memory model.

B.3.4 Creating the LIBATTR Module

Each SRTL is associated with a specific set of attributes. These attributes include:

- the SRTL's name
- the SRTL's version number
- the SRTL's data location when using the Small memory model
- whether the SRTL is shared or not shared by default

The attributes of a SRTL are established by including a module with the name "LIBATTR" in the SRTL library file.

Note: The LIBATTR module must be specified as the first module in the list.

The LIBATTR module should contain only a single data segment with name "LIBATTR". The contents of this segment must have the format indicated by the lib_id and lib_attr structures shown in the listing below. The equivalent form in RASM-86 code appears in the example at the end of this appendix.

```
struct lib_id {
        char            li_name[8];
        unsigned short  li_ver_major;
        unsigned short  li_ver_minor;
        unsigned char   li_flags[4];
};
struct lib_attr {
        lib_id          la_id;
        unsigned short  la_data_offset;
        char            la_share_attr;
};
#define LA_S_SHARED     0x1
```

LIB_ID Structure

The lib_id structure defines the fields used to identify the SRTL including its name, version numbers and flags. The lib_id fields are:

li_name — This is the physical name of the XSRTL specified by users of the SRTL. If this field is nonblank, the library is a SRTL; otherwise the library is a normal library. This field must be exactly 8 bytes long, including trailing blanks.

li_ver_major — This is the major version number. It should be updated when there are major changes to a library that make it incompatible with previous versions. The FlexOS program loader verifies that the major version number specified in a user program is identical to that of the XSRTL.

li_ver_minor — This is the minor version number. It should be updated when changes to the library do not create incompatibilities with previous versions. The minor version number specified in a user program does not have to exactly match that specified in the XSRTL.

li_flags — This field contains flags that distinguish different variants of the library from one another. Currently, only the low-order 6 bits of li_flags[3] have been assigned the following values:

LI_F_CMD (0x0)	This flag informs LINK 86 the code file suffix should be CMD. If this, or any other value is specified to a version of LINK 86 that cannot generate that style of code file, an error is generated. For example, the LINK 86 version that generates CMD and 286 files cannot generate an EXE file and vice versa.
LI_F_286 (0x1)	This flag informs LINK 86 the code file suffix should be 286.
LI_F_EXE (0x2)	This flag informs LINK 86 the code file suffix should be EXE.
(0x3 through 0xF)	These values are unassigned and cause an error if specified in a LIBATTR module.
LI_F_DUP_MAIN (0x10)	This value informs LINK 86 that duplicate definitions of the symbol "main" are permitted and should not generate errors.
LI_F_DS_STACK (0x20)	This value informs LINK 86 the stack appears at the low end of the data segment. When this bit is set, the value of the data_offset field represents the size of this stack. If multiple LIBATTR modules are found, the smallest data_offset value is used. Data is allocated above the stack and the public variable ?STACK, if present, is initialized to the size of the stack. The variable ?STACK is assumed to be allocated a word (2 bytes).

LIB_ATTR Structure

The fields in the lib_attr structure determine the SRTL's memory offset, and whether or not the SRTL is shareable. The lib_attr fields are:

la_id	This field specifies the library id.
la_data_offset	With the Small memory model, this field contains the fixed address where the SRTL data must appear. This value is also used to prevent LINK 86 from allocating user data at the same location as the SRTL data. With the Large memory model, this field must have the value 0xFFFF.
LA_SHARE_ATTR	This field tells LINK 86 whether the library by default is shared (value 1) or not shared (value 0), so you do not have to specify the SHARED option every time the library is used. You can override this default as specified in Section 7.5.

When linking a user program, LINK 86 determines that a library is a SRTL by checking the name of the first module in a library. If the first module has the name "LIBATTR", LINK 86 examines the contents of the LIBATTR segment to determine the attributes of the library. Depending on the value of the LA_SHARE_ATTR field and input options, the library is treated either as a SRTL or as a regular library.

Before creating a LIBATTR module, you must determine the filename of the executable shareable runtime library (see Section B.3.6) and the data offset if the Small memory model is used. You must also ensure the sizes and offsets of data items correspond to the fields in the LIBATTR definition given above. The LIBATTR segment must contain exactly 19 bytes.

When linking your file(s) with a SRTL, do not include the LIBATTR module in your user code file. The LIBATTR module is used only to define the attributes of the SRTL and should not be treated as a normal module. The LIBATTR module is included in the code file only when the SRTL is linked as a normal, nonshared library (without specifying the SEARCH option).

B.3.5 Creating the SRTL

You create a SRTL using LIB-86. When entering the command line, you must specify the LIBATTR object file as the first file in the library. If the optional SRTL Transfer Vector is used, it must be the second object file in the library.

For example, to create a SRTL named SM1SRTL using the LIBATTR file, LIBATTR.OBJ, and the object files: TVECT.OBJ, SVCIF.OBJ, and RTESRTL.OBJ enter the command:

```
lib86 sm1srtl.186[map,xref]=libattr.obj,tvect.obj,svcif.obj,rtesrtl.obj
```

Assuming all calling conventions are correct, you can modify an existing nonshareable runtime library to create a shareable library by including a LIBATTR file in the LIB86 command. For example, to make the nonshareable library OLDLIB.L86 into the shareable library NEWLIB using a LIBATTR file named LIBATTR.OBJ, enter:

```
lib86 newlib = libattr.obj, oldlib.186
```

B.3.6 Creating the XSRTL

A library file cannot be loaded by the operating system and executed. Therefore, an executable version of the library file must be available. This version of a SRTL is called the XSRTL (eXecutable Shared Run Time Library).

Once you create a SRTL, you then create the XSRTL by linking the SRTL, as shown in the following example. LINK 86 automatically recognizes the library is a SRTL by the presence of the LIBATTR module.

The command:

```
link86 sm1srtl.srl=sm1srtl.186
```

creates a file SM1STRL.SRL, containing the executable version of the library file SM1SRTL.L86. With the exception of the LIBATTR module, LINK 86 processes all code and data from all modules in the library in the order they appear in the library. During the link process, the addresses of the data are resolved, but the data is not included in the XSRTL, which contains only the SRTL code. This convention is necessary to ensure that the references in a SRTL user program match the XSRTL, since both are linked separately.

The XSRTL has no main program, but this is not a problem, because LINK 86 knows it is an XSRTL. Also, the XSRTL cannot contain any unresolved external references, because there is no way to resolve them separately to each of multiple, simultaneous user programs.

Note that the value of the data_offset attribute from the LIBATTR module determines whether the XSRTL uses the Small or Large memory model. If the attribute value is 0xFFFF, the Large memory model is assumed. Any other value indicates the offset of the SRTL data within the data segment, which means the XSRTL uses the Small memory model.

B.4 Small & Medium Model SRTLs

When creating a SRTL for use with the Small or Medium Memory Model, you must decide beforehand where the SRTL data appears and code this as the value of the data_offset attribute in the LIBATTR module. The location DS:0 is not acceptable, because the stack overflow detection requires the stack to be at the bottom of the data segment. On the other hand, putting the SRTL data at the top of the data segment does not efficiently utilize memory. The layout of the data segment for a Small memory model program should resemble that specified in Figure B-1.

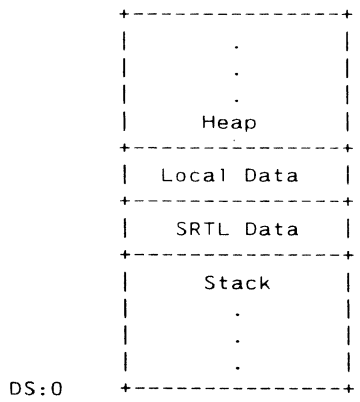


Figure B-1. Small Memory Model Data Segment

If the size of the stack exceeds the starting address of the SRTL data, LINK 86 prints an error message and terminates the link.

B.4.1 Calling Conventions

A file compiled using the Small memory model must be linked with a Small model SRTL so that both the user file and the SRTL can agree on the location of the SRTL data. This restriction fixes the calling convention as illustrated in Figure B-2 below.

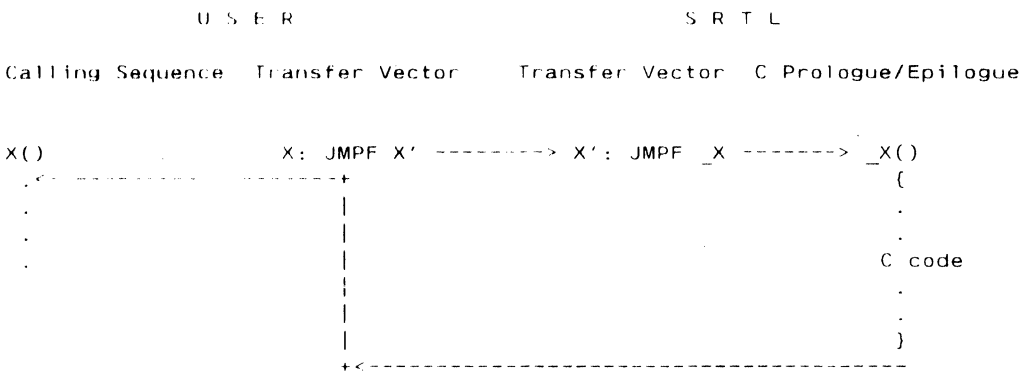


Figure B-2. Small Memory Model Calling Convention

The calling convention shown in Figure B-2 uses a SRTL Transfer Vector. If no SRTL Transfer Vector is used, the reference to the SRTL routine would go directly to the SRTL instruction, rather than the JMPF instruction in the transfer vector.

Note that the SRTL code sequence in Figure B-2 can have one or more code segments and, therefore, can be used with the standard Medium memory model consisting of multiple code segments and a single data segment.

B.5 Large Model SRTLs

To have reentrant code, the SRTL data must belong to the user file, rather than the SRTL. When using the Large memory model, the user file must inform the SRTL of the location of its data. As a consequence, user calls to Large memory model SRTL routines must pass through a User Transfer Vector, as described in Section B.3.2.

B.5.1 Calling Conventions

The SRTL determines the location of the user file's data by mapping all references to the SRTL through a transfer vector local to the user. The transfer vector pushes the user's DS and loads the SRTL DS before calling the SRTL routine, then pops the user's DS after control returns from the library. This convention is shown in Figure B-3.

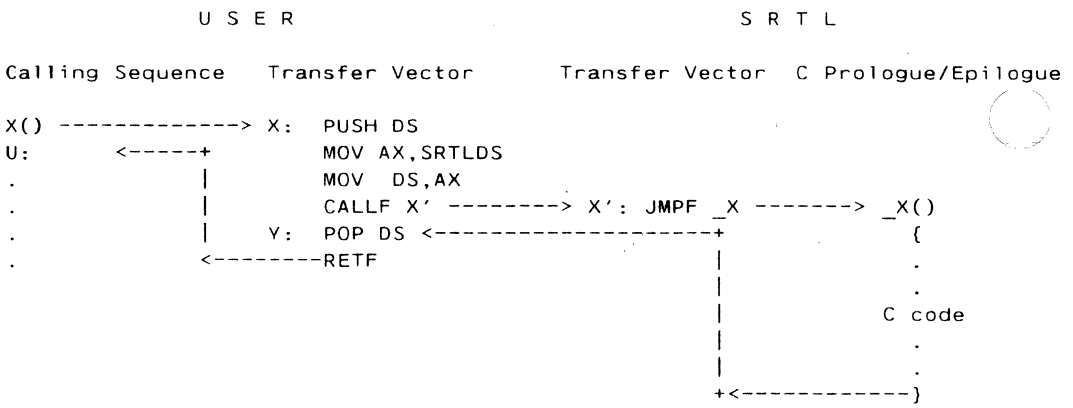


Figure B-3. Large Memory Model Calling Convention

Note that this calling convention assumes the creator of the library coded a transfer vector containing the entry points to the library. If this were not the case, the CALLF in the user's transfer vector would directly reference the instruction in the SRTL routine. The transfer vector on the SRTL side, though not necessary, is recommended.

B.6 SRTL Restrictions

When creating SRTLs, keep the following restrictions in mind:

- XSRTLs cannot contain any unresolved external references.
- XSRTLs cannot contain overlays.
- A SRTL can contain a maximum of 255 object modules, not counting the LIBATTR module.

B.7 Example SRTL

The following C code tests a Small memory model SRTL by calling the SRTL subroutine LIST_PRINT twice.

```
/*************************************************************************
 *    Main program to call a Shared Run Time Library (SRTL)              *
 *************************************************************************/

extern _far void list_print( _far char *  );

void main()
{
    list_print( ( far char *)"\n\r\tIn Main first time : " );
    list_print( ( far char *)"\n\r\tIn Main second time: " );
}
```

The C code on the following page defines the LIBATTR (Library Attribute) module. The LIBATTR module defines the attributes used by LINK 86 when linking XSRTLs (eXecutable SRTLs) and when linking to other SRTLs. The LIBATTR module must appear as the first module in a SRTL.

```
/*-----------------------------------------------------------------*
 *      LIBATTR.C  -- The library attribute module for a SRTL      *
 *-----------------------------------------------------------------*/

typedef struct
    {
        char            li_name[8];
        unsigned short  li_ver_major;
        unsigned short  li_ver_minor;
        unsigned char   li_flags[4];
    } lib_id;

typedef struct
    {
        lib_id          la_id;
        unsigned short  la_data_offset;
        char            la_share_attr;
    } lib_attr;

static  lib_attr  attrib =
    {
        {               /* -------- Lib_id structure --------*/
            "SM1SRTL ",         /*  li_name[8]               */
            0x9876,             /*  li_ver_major             */
            0x5432,             /*  li_ver_minor             */
            { 0,0,0,0x31 }      /*  li_flags[4]              */
        },              /*-----------------------------------*/
        0x0400,                 /*  la_data_offset           */
                                /*   0xFFFF for large model  */
        01                      /*  Shared attribute         */
    };

/*--------------------------End of LIBATTR.C-----------------------*/
```

The C code on the following page defines SRTL routines that test Small memory model SRTLs. The LIST_PRINT routine loads the string into the variable I and calls the CONSOLE_PRINT, which prints the string on the console.

```
/*
 *       'C' Shared Run Time Library routines
 */
#include "flextab.h"

/*************************************************************************
 *    _FILL_CHAR is a MetaWare intrinsic function                        *
 *************************************************************************/
#define blkfill(where,what,how_many) _fill_char(where,how_many,what)
#define ZERO_PARMS blkfill(&pblock,NULL,sizeof(pblock))

extern       long __svcif();
       _far void _listprint( _far  char * );
            long console_print( _far char * );

/*************************************************************************
 *    _LISTPRINT                                                         *
 *       This is the entry into the Shared Run Time Library for the      *
 *       sample program TEST.C                                           *
 *************************************************************************/
_far void _listprint( _far char *i )
{
    console_print( (_far char *)"\n\rNow in the SRTL XX    " );
    console_print( i );
}
```

```
/*********************************************************************
 *   CONSOLE_PRINT                                                   *
 *      This is the same as a runtime library call to the S_WRITE    *
 *      function.  We must use it here because we can not use any    *
 *      external references outside of our Shared Run Time Library.  *
 *********************************************************************/
long console_print( _far char *buffer )
{
        struct  PARAMBLOCK pblock;

        ZERO_PARMS;                     /* zero-init entire parm block */

        pblock._pflags = 0;
        pblock._pparam1 = 1L;           /* Fnum for STDOUT      */
        pblock._pparam2._pp2long = (LONG)buffer ;
        pblock._pparam3 = 24L;          /* Length of the Write. */
        pblock._pparam4 = 0L;

        return( __svcif( F_WRITE, (_far BYTE *)&pblock ) );
}
/*--------------------------End of RTESRTL.C-----------------------*/
```

End of Appendix B

LINK 86 Error Messages

During the course of operation, LINK 86 can display error messages. The error messages and a brief explanation of their cause are listed below. The number in parenthesis following the error message is the system return code. This number can be used with the IF command in a batch file to determine if an error condition is true of false.

Table H-1. LINK 86 Error Messages

Message	Meaning
NO BLOCK OR DIRECTORY ENTRIES AVAILABLE (01)	There is no more space available on the disk for data or directory entries.
8087 IN OVERLAY, NOT IN ROOT (02)	The 8087 emulator, if used, must be referenced in the root if it is to be referenced in an overlay.
8087 SWITCH OCCURRED AFTER FIRST FILENAME (03)	The HARD8087, AUTO8087, and SIM8087 switches must not appear after the first object file listed on the command line.
8087 TABLE OVERFLOW (04)	The 8087 fixup table needed with the AUTO8087 or SIM8087 options can have a maximum size of 64K.
ALIGN TYPE NOT IMPLEMENTED (05)	The object file contains a segment align type not implemented in LINK 86.

Table H-1. (continued)

Message	Meaning
CANNOT CLOSE (06)	LINK 86 cannot close an output file. Check to see if the correct disk is in the drive and the disk is not write-protected or full.
CLASS NOT FOUND (07)	The class name specified in the command line does not appear in any of the files linked.
COMBINE TYPE NOT IMPLEMENTED (08)	The object file contains a segment align type not implemented in LINK 86.
COMMAND TOO LONG (09)	The total length of input to LINK 86, including the input file, cannot exceed 2048 characters.
DISK READ ERROR (11)	LINK 86 cannot properly read a source or object file. This is usually the result of an unexpected end-of-file character. Correct the problem in your source file.
DISK WRITE ERROR (12)	A file cannot be written properly; the disk is probably full.
ERROR IN LIBATTR MODULE (13)	The LIBATTR module does not conform to established requirements. Fix the LIBATTR module and rebuild the library in question.

Table H-1. (continued)

Message	Meaning
FIXUP TYPE NOT IMPLEMENTED (14)	The object file uses a fixup type not implemented in LINK 86. Make sure the object file has not been corrupted.
GROUP NOT FOUND (15)	The group name specified in the command line does not appear in any of the files linked.
GROUP OVER 64K (16)	The group listed must be made smaller than 64k before relinking. Either delete segments from the group, split it up into 2 or more groups or do not use groups.
GROUP TYPE NOT IMPLEMENTED (17)	LINK 86 only supports segments as elements of a group.
INVALID LIBRARY-REQUESTED SUFFIX (18)	The command file suffix requested by a library is not supported. Verify that the correct library is being used.
LINK-86 ERROR 1 (19)	This error indicates an inconsistency in the LINK 86 internal tables, and should never be emitted.
MULTIPLE DEFINITION (20)	The indicated symbol is defined as PUBLIC in more than one module. Correct the problem in the source file, and try again.
MORE THAN ONE MAIN PROGRAM (21)	A program linked by LINK 86 may have at most one main program.

Table H-1. (continued)

Message	Meaning
NO FILE (22)	LINK 86 cannot find the indicated source or object file on the indicated drive.
OBJECT FILE ERROR (23)	LINK 86 detected an error in the object file. This is caused by a translator error or by a bad disk file. Try regenerating the file.
RECORD TYPE NOT IMPLEMENTED (24)	The object file contains a record type not implemented in LINK 86. Make sure the object file has not been corrupted by regenerating it and linking again.
SEGMENT OVER 64K (25)	The segment listed after the error message has a total length greater than 64k bytes. Make the segment smaller, or do not combine it with other PUBLIC segments of the same name.
STACK COLLIDES WITH SRTL DATA (26)	The base address of SRTL data does not allow enough room for the requested amount of stack space. Change the base of the SRTL data in the LIBATTR module or request less stack.
SRTL DATA OVERLAP (27)	The data from 2 SRTLs overlap. Change the base address in the LIBATTR module of one of the SRTLs.
SRTL CANNOT CONTAIN 8087 FIXUPS (28)	A SRTL cannot use the 8087 emulator as currently implemented.

Table H-1. (continued)

Message	Meaning
SEGMENT CLASS ERROR (29)	The class of a segment must be CODE, DATA, STACK, EXTRA, X1, X2, X3, or X4.
SEGMENT ATTRIBUTE ERROR (30)	The Combine type of the indicated segment is not the same as the type of the segment in a previously linked file. Regenerate the object file after changing the segment attributes as needed.
SEGMENT COMBINATION ERROR (31)	An attempt is made to combine segments that cannot be combined, such as LOCAL segments. Change the segment attributes and relink.
SEGMENT NOT FOUND (32)	The segment name specified in the command line does not appear in any of the files linked.
SYMBOL TABLE OVERFLOW (33)	LINK 86 ran out of Symbol Table space. Either reduce the number or length of symbols in the program, or relink on a system with more memory.
SYNTAX ERROR (34)	LINK 86 detected a syntax error in the command line; the error is probably an improper filename or an invalid command option. LINK 86 echoes the command line up to the point where it found the error. Retype the command line or edit the INP file.
TARGET OUT OF RANGE (35)	The target of a fixup cannot be reached from the location of the fixup.

Table H-1. (continued)

Message	Meaning
TOO MANY MODULES IN LIBRARY (36)	A library cannot contain more than 512 modules. Split the library into 2 or more libraries and relink.
TOO MANY MODULES LINKED FROM LIBRARY (37)	A library cannot supply more than 512 modules during the linkage process. Split the library into 2 or more smaller libraries and relink.
UNDEFINED SYMBOLS (38)	The symbols following this message are referenced but not defined in any of the modules being linked.
XSRTL MUST BE LINKED BY ITSELF (40)	When linking an XSRTL, no other files may be linked at the same time.
XSRTLs INCOMPATIBLE WITH OVERLAYS (41)	An XSRTL cannot use overlays.
OBJECT FILE OVER 64K (42)	The object file created by LINK 86 is greater than 64k. Break the code into modules and relink.
TOO MANY MODULE NAMES (43)	There are too many library module names. Combine modules and relink.

End of Appendix H

The pages following this one are change pages for the FlexOS Programmer's Guide, First Edition November 1986 (1073-2024-001).

Replace the following pages:

1-3, 1-4

3-11, 3-12, 3-19, 3-20, 3-21, 3-22

7-7, 7-8, 7-9, 7-10, 7-15, 7-16, 7-57, 7-58, 7-59, 7-60, 7-63, 7-64, 7-71, 7-72, 7-85, 7-86, 7-87, 7-88, 7-117a, 7-117b, 7-118

8-1, 8-2, 8-5, 8-6, 8-9, 8-10, 8-23, 8-24, 8-24a, 8-24b, 8-25, 8-26, 8-27, 8-28, 8-43, 8-44, 8-45, 8-46

B-3, B-4, B-5, B-6, B-7, B-8, B-9, B-10

Table 1-2. (Continued)

Purpose	Call	Action
Real Time and Process Management		
	TIMER*	Set and wait for timer interrupt
	ABORT*	Abort specified process
	COMMAND*	Perform command
	EXCEPTION	Set software interrupts on exceptions
	MALLOC	Allocate memory to heap
	MFREE	Free memory from heap
	EXIT	Terminate with return code
	ENABLE	Enable software interrupts
	DISABLE	Disable software interrupts
	SWIRET	Return from software interrupt
	CONTROL*	Control a process for debugging
	OVERLAY	Load overlay from command file
Device Management		
	SPECIAL*	Perform special device function
	DEVLOCK	Lock or unlock device for user/group
	INSTALL	Install, replace and associate drivers
Table Management		
	GET	Get a table
	SET	Set table values
	LOOKUP	Scan and retrieve tables

* Your program can call these SVCs asynchronously.

Table 1-3 lists the SVCs by their number.

Table 1-3. Supervisor Calls by Number

Number	Call	Number	Call
0	F_GET	21	Reserved
1	F_SET	22	F_GIVE
2	F_LOOKUP	23	F_BWAIT
3	F_CREATE	24	F_TIMER
4	F_DELETE	25	F_EXIT
5	F_OPEN	26	F_ABORT
6	F_CLOSE	27	F_CANCEL
7	F_READ	28	F_WAIT
8	F_WRITE	29	F_STATUS
9	F_SPECIAL	30	F_RETURN
10	F_RENAME	31	F_EXCEPTION
11	F_DEFINE	32	F_ENABLE
12	F_DEVLOCK	33	F_DISABLE
13	F_INSTALL	34	F_SWIRET
14	F_LOCK	35	F_MALLOC
15	F_COPY	36	F_MFREE
16	F_ALTER	37	F_OVERLAY
17	F_XLAT	38	F_COMMAND
18	F_RWAIT	39	F_CONTROL
19	F_KCTRL	40	F_GSX
20	F_ORDER	41	F_SEEK

1.2.1 Calling Conventions

FlexOS Supervisor calls are made by invoking the FlexOS entry point. The entry point takes two arguments and returns a value, as follows:

Arguments: a SVC 16-bit number
a parameter block pointer or value, 32-bit

Return: a 32-bit value

RECT C Structure

The RECT data structure defines a rectangular region of a FRAME. The point of reference is the FRAME coordinates of the region's upper lefthand corner. The region's width and height are specified within the data structure in terms of character rows and columns. The SVCs using the RECT structure specify which FRAME planes are included in the RECT. Figure 3-5 shows the RECT data structure diagram. The corresponding C structure is as follows:

```
struct RECT
{
        WORD    row,col,nrow,ncol;
/*              Top left corner FRAME coordinates
                and RECT width and height*/
}
```

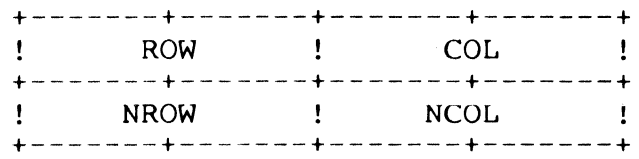


Figure 3-5. RECT Structure

The RECT fields are defined as follows:

- **row**: The row coordinate relative to the FRAME of the rectangle's upper lefthand corner (counting begins at row 0)
- **col**: The column coordinate relative to the FRAME of the rectangle's upper lefthand corner (counting begins at row 0)
- **nrow**: The number of rows (height) in the rectangle
- **ncol**: The number of columns (width) in the rectangle

3.2 Controlling the Console

Console attributes such as screen and keyboard modes, cursor location, and the number of character rows and columns are contained in the CONSOLE table. You manage the console screen on a FRAME basis with the ALTER and COPY SVCs and on a character basis with the WRITE SVC.

3.2.1 Console Attributes

The CONSOLE table is your reference source for information regarding console attributes and conditions. Figure 3-6 illustrates the CONSOLE table data structure. To get or set your process's CONSOLE table, use 0 or 1 or the stdin and stdout file numbers from the ENVIRON table as the GET or SET ID value

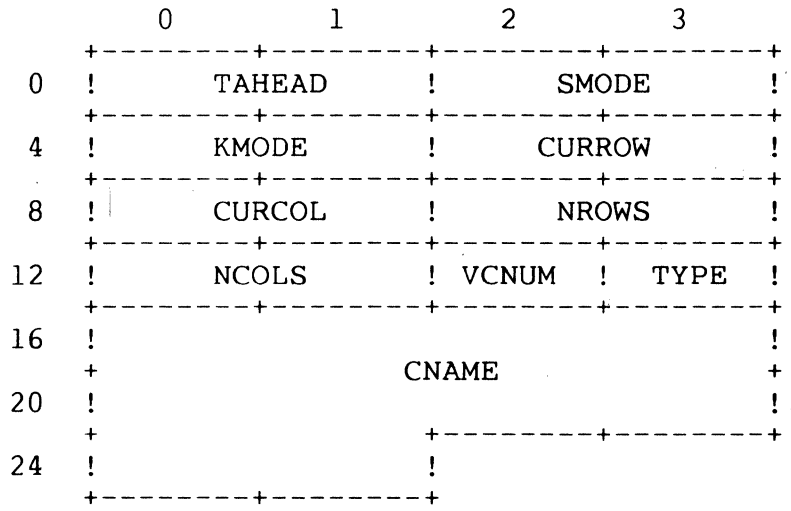


Figure 3-6. CONSOLE Table

Opening the Mouse File

The mouse is opened by calling OPEN. In your OPEN call you specify the mouse name, the access privileges required, and the access mode. The mouse name is vcxxx/mouse where xxx is a decimal number indicating the current virtual console number. Get the virtual console number from the VCNUM field in your standard input file's CONSOLE table. (Call GET with an ID value of 0 to retrieve stdin's CONSOLE table.) For example, if the VCNUM value is 3, your mouse name would be vc003/mouse.

In your OPEN call, specify at least read access privilege. If you need to set the MOUSE table, request set access as well. For the access mode specify exclusive mode unless mouse access will be shared by multiple processes. In this case, specify shared, shared file pointer mode. Access is restricted to processes with the same family ID.

Your application should close the mouse file when you are done, otherwise you cannot close or delete the virtual console. CLOSE flag bit 0 has no meaning with respect to the mouse and is ignored.

Using BWAIT

Use the BWAIT SVC to monitor button state changes. BWAIT counts the number of times a specified mouse button condition occurs within a given time period. A button condition is defined by two criteria: buttons and their ON or OFF state.

The BWAIT form is as follows:

```
  ret = s_bwait(clicks,fnum,mask,state);
emask = e_bwait(swi,clicks,fnum,mask,state);
```

The fnum value is the file number returned when you OPEN the vcxxx/mouse file. The mask and state parameters are 32-bit values which define the mouse button condition.

You select buttons for the mask value by their position on the mouse. The leftmost button is represented by the least significant bit in the mask; the next button to the left is represented by the next bit, and so forth. To select the button, set its corresponding mask bit.

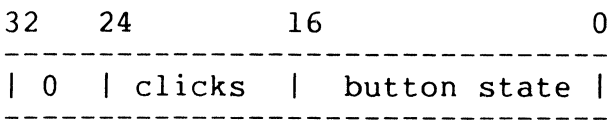
You define whether the button selected is to be ON or OFF in the state value. The Console Resource Manager looks only at the state bits corresponding to the buttons selected in the mask. Set the bit for ON.

As an example of the use of the mask and state fields, consider a two-button mouse. You can have the following button conditions:

1. The left button is pressed (ON) without concern for the state of any other buttons: mask = 1, state = 1.
2. The left button is pressed while the right button is not: mask = 3, state = 1.
3. The right button is pressed while the left button is not: mask = 3, state = 2.
4. The right button is pressed without concern for the state of any other buttons: mask = 2, state = 2.
5. If either buttons is pressed: mask = 3, state = 3.

Use the clicks value to delimit the event by a specific number of incidences of the specified button condition. You can specify any number of clicks between 0 and 255. Use a click value of 0 to determine the mouse's current condition. BWAIT returns with a value of 0 when you specify 0 clicks and the mouse is in the condition defined in the mask and state.

BWAIT counts button conditions for a limited time period--the CLICK time limit specified in the MOUSE table. If the time period expires before the BWAIT click count is reached, the event terminates. The Console Resource Manager starts the timer upon the first incidence of the condition. Consequently, the count returned is always at least one except as described above. BWAIT returns a LONG value containing the following:



Using RWAIT

RWAIT establishes an event boundary for the mouse. RWAIT returns with the row and column coordinates of the mouse's hotspot when it crosses the boundary. The RWAIT form is as follows:

```
position = s_rwait(flags,fnum,region);
   emask = e_rwait(swi,flags,fnum,region);
```

Set RWAIT flag bit 0 to clip the region to the current window borders. Otherwise, the region can include areas not visible on the parent screen. Flag bit 1 determines if the event occurs when the form exits or enters the region. Flag bit 2 allows the region rectangle to be defined as: COL,ROW,NCOLS,NROWS instead of the default order:ROW,COL,NROWS,NCOLS. The other flag bits are not used.

The region is a RECT structure confined to the calling process's virtual console's FRAME. The position value returned is 32-bits where the high order word indicates the row and the low order word the column.

3.4 Managing Virtual Consoles

For applications with multiple processes sharing access to the console and keyboard, it is often necessary or convenient to have a separate virtual console for each process. The key to these applications is a process--the window manager--which creates the virtual consoles, sets each window's size and position, and passes keyboard and mouse access from one process to another according to a planned transfer scheme. (These are basically the same functions as the FlexOS window manager supplied with the operating system.)

The window manager flow chart would include the following FlexOS functions; the SVCs used appear in parentheses.

1. Create a virtual console (CREATE).
2. Get the virtual console number (GET).
3. Set the virtual console's window size and location (SET).
4. Assign the console file to stdin, stdout, and stderr (DEFINE).
5. Define conditions under which keyboard and mouse ownership is returned (KCTRL and/or MCTRL).

6. Invoke shell or application that will use screen (asynchronous COMMAND).
7. Give keyboard and mouse ownership to the new virtual console (GIVE).
8. Read from your keyboard buffer (READ).
9. Reorder the virtual consoles to put the selected one on top (ORDER).

Steps 1 through 5 are repeated to create each virtual console. You have a numerical limit of 255 virtual consoles.

3.4.1 Creating the Virtual Consoles and Windows

To create a virtual console, you must specify the console screen on which it is to appear. This is called the parent screen. The virtual console created is referred to as a child console. Child consoles created on the same parent screen are referred to as sibling consoles. There are four rules of virtual console management based on these relationships:

- A child console always overlays its parent.
- Sibling consoles are "stacked" on the parent in the order of their creation until reordered by ORDER.
- The ORDER SVC only reorders a "stack" of sibling virtual consoles and cannot be used to put a parent on top of a child.
- An application always has access to its entire console regardless of its virtual console's position in the stack and the size of its window.

Figure 3-9 illustrates the parent, child, and sibling console relationships and the three rules. As shown in this figure, you can have multiple tiers of virtual consoles. As you change tiers, the parent/child relationships change. All virtual consoles on a given level are siblings.

7.3 BWAIT

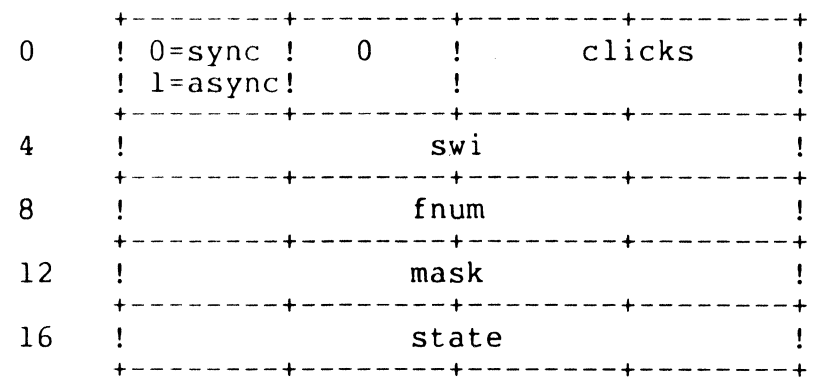
C Interface:

```
UWORD       clicks;
LONG        mask;
LONG        state;

ret = s_bwait(clicks,fnum,mask,state);
emask = e_bwait(swi,clicks,fnum,mask,state);

ret = __osif(F_BWAIT,&parmblk);
```

parmblk:



Parameters:

clicks — Number of times the mouse enters this state within the "click interval" set up in the MOUSE Table after this call is made. If **clicks** is 0 and the mouse is already in this state, the event is already complete. A maximum number of 255 clicks is allowed.

fnum — Mouse file number

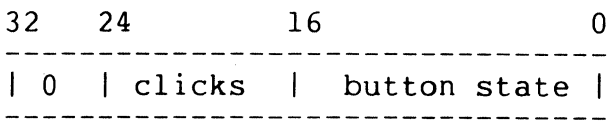
mask — Bit mask of buttons to consider. The lowest order bit is set if the first mouse button to the left is to be considered. The second lowest bit corresponds to the second button from the left. A total of 16 mouse buttons can be supported in the low word of **mask**.

state — Bit mask of buttons that define the button state given the mask that determines the buttons to ignore all together in the low word of **state**. The event will complete on any of the following conditions:

- when the number of clicks is satisfied
- when the number of clicks is not satisfied but the timer interval elapses (see MOUSE Table, in Section 8).
- when the number of clicks = 0 and the button is in the state requested

Return Code:

ret



Error Code

Description: The BWAIT SVC allows the calling process to wait until a mouse button state is reached. The **mask** determines the number of mouse buttons the calling process wants considered. For example, by setting the mask appropriately, a one button mouse can be expected when there is more than one button.

The **clicks** field allows the calling process to receive multi-click mouse input. When a user presses a mouse button, releases it and presses it again within the "click interval", the mouse has been double clicked.

If **clicks** is set to two, and a second click is not performed within the "click interval", the event is considered complete. The return value indicates the number of clicks actually performed and the last valid state of the buttons at the time of completion. If **clicks** is set to zero, BWAIT returns a zero if the button state is already in the specified state. Otherwise, it returns one upon the first entry to the state.

The "click interval" is changed in the MOUSE table through the SET SVC.

7.4 CANCEL

C Interface:

```
LONG        dmask;
LONG        events;

dmask = s_cancel(events);

dmask = __osif(F_CANCEL,events);
```

Parameters:

events	Logical OR of event masks to be canceled

Return Code:

dmask	Bit map of events that could not be canceled because they have already completed

Description: The CANCEL SVC terminates one or more specified asynchronous SVCs. The events argument is the logical OR of the event masks you want to cancel. The dmask return code indicates events that, although requested for termination, had already completed. Use the RETURN SVC to get the return codes for these events so the event bits can be reused.

Parameters:

flags bits 0-3 are reserved

bit 4: 1 = No new process (set bit 5 to 1)
0 = New process (ignore bit 5)

bit 5: 1 = Chain
0 = Not implemented (returns E_IMPLEMENT error)

bit 6: reserved (must be 0)

bit 7: 1 = Assign a new process family ID (FID)
0 = Keep the current process family ID (FID)

bits 8-12 are reserved (must be 0).

bit 13: 1 = Force case to media default
0 = Do not affect name case

bit 14: 1 = Literal command
0 = Prefix substitution allowed

bit 15: reserved (must be 0)

swi Address of a software interrupt routine

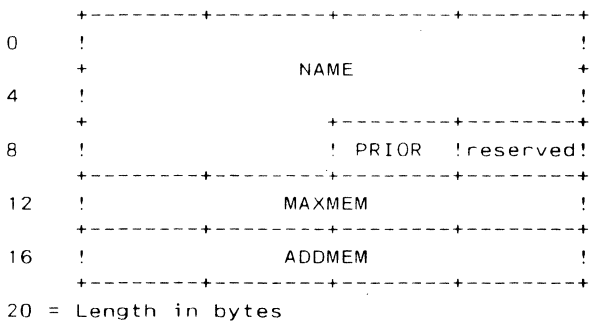
command Address of 128-byte, null-terminated string indicating the name of the loadable file.

buffer Address of a variable length buffer containing a 128-byte, null-terminated command tail and special information to be passed to the new process. (At most, the command tail can be 127 characters and one NULL byte long.) COMMAND puts the tail in the CMDENV table. Data after the first 128 bytes is put in the process's heap.

The PROCESS table contains the heap address and size. Use this buffer area to pass an environment string, common data, or special information to the program.

bufsize — Size of buffer in bytes

procinfo — Address of the PINFO table. PINFO must be constructed as follows:



name: Process name

prior: Process priority (user processes are usually set to 200)

maxmem: Maximum memory this process can own (larger minimum requirements specified by the command file supercede this amount)

addmem: The amount of memory to be added to the minimum amount specified by the command file (FlexOS allocates the greater of the two values: maxmem or the sum of the command file's specified minimum plus addmem)

pid — Address of new process ID. COMMAND puts the new process's 32-bit PID at this location when flag bit 4 equals 0 and COMMAND is called asynchronously.

7.21 KCTRL

C Interface:

```
LONG        fnum;
UWORD       nranges;
UWORD       flags,beg1,beg2,beg3,beg4;
UWORD       end1,end2,end3,end4;
RECT        region;

ret = s_kctrl(fnum,nranges,beg1,end1,beg2,end2,...end4);
ret = s_mctrl(fnum,region);
ret = s_gmctrl(fnum,region);
ret = __osif(F_KCTRL,&parmblk);
```

parmblk:

```
    +--------+--------+--------+--------+
0   !   0    !   0    !       flags     !
    +--------+--------+--------+--------+
4   !                 0                 !
    +--------+--------+--------+--------+
8   !                fnum               !
    +--------+--------+--------+--------+
12  !       beg1     !       end1       !
    +--------+--------+--------+--------+
16  !       beg2     !       end2       !
    +--------+--------+--------+--------+
20  !       beg3     !       end3       !
    +--------+--------+--------+--------+
24  !       beg4     !       end4       !
    +--------+--------+--------+--------+
          (if mouse control)
    +--------+--------+--------+--------+
12  !        ROW     !        COL       !
    +--------+--------+--------+--------+
16  !       NROWS    !       NCOLS      !
    +--------+--------+--------+--------+
```

Parameters:

flags

bit 0: 1 = Mouse control
0 = Character control

bit 2: 1 = default region definition
(ROW,COL,NROWS,NCOLS)

0 = GEM-compatible region definition
(COL,ROW,NCOLS,NROWS)

If bit 0 = 0, keyboard and mouse ownership is controlled through characters typed on the keyboard and the begin range and end range parameters are required. If bit 0 =1, keyboard and mouse ownership is controlled through mouse movement and a region is required.

If bit 2 = 0, the region is GEM-like. If bit 2 =1, the region has the default definition.

nranges — The number of beginning and ending ranges to follow--maximum 4.

fnum — Console file number of console to get keyboard; must be console file of the parent virtual console.

begn — First character in range of characters; pressing any character in range causes keyboard to return to specified console.

endn — Last character in the range.

region — RECT structure defining a character rectangle on the parent's virtual console.

Return Code:

ret	Error Code

Description: The KCTRL SVC transfers keyboard ownership to the console file specified by fnum when a character is entered that falls within any of the four ranges specified. The initial transfer of ownership is conferred with the GIVE SVC.

You can specify up to four character ranges. The ranges are inclusive of the first and last characters. A single character is specified by using it as the beginning and ending character. When a character falling in the range is typed, that character and all subsequent characters are diverted to the parent console file's keyboard buffer. The process controlling the virtual consoles can either give control of the keyboard to another virtual console or take some special action on behalf of the user.

You can also use mouse position to change keyboard and mouse ownership. In this case you specify a RECT (see Section 3 for the RECT description) on the parent console in which the mouse form must be resident. This region must be within the virtual console. When the mouse leaves the region, keyboard and mouse ownership go back to the parent. This happens as long as the rectangle's size is greater than 0. The parent's application must set NROWS and NCOLS to disable a previously defined s_mctrl if it wants to regain ownership whenever the mouse is outside the specified rectangle.

s_kctrl and s_mctrl are disabled by reversing the order of the beginning and ending character ranges, or providing a rectangle of size 0.

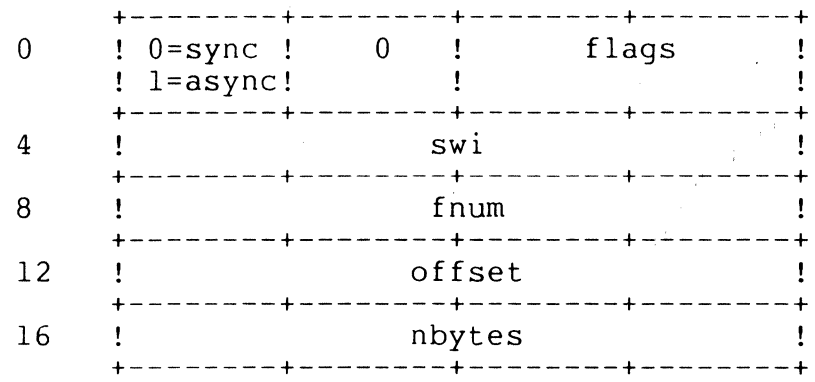
7.22 LOCK

C Interface:

```
UWORD       flags;
LONG        fnum,offset,nbytes;

ret = s_lock(flags,fnum,offset,nbytes);
emask = e_lock(swi,flags,fnum,offset,nbytes);
ret = __osif(F_LOCK,&parmblk);
```

parmblk:



Parameters:

flags — bits 0 and 1 select the LOCK mode

- 0 = Unlock
- 1 = Exclusive lock
- 2 = Exclusive write lock
- 3 = Shared write lock

bits 2-3 are reserved (must be 0)

7.23 LOOKUP

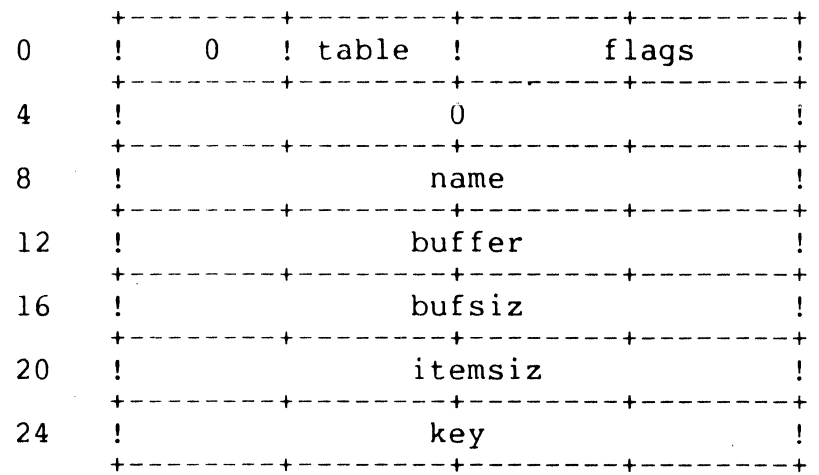
C Interface:

```
UWORD       flags;
BYTE        table,*name,*buffer;
LONG        key,bufsiz,itemsiz,nfound;

nfound = s_lookup(table,flags,name,buffer,bufsiz,itemsiz,key);

ret = __osif(F_LOOKUP,&parmblk);
```

parmblk:



Parameters:

table	Table Number (Table 10-1 lists the table numbers)
flags	bits 0 - 7 are dependent on table type
	bits 8 -12 are reserved (must be 0)

bit 13: 1 = Force name case to media default
0 = Do not change name case

bit 14: 1 = Literal name
0 = Prefix translation allowed

bit 15 is reserved (must be 0)

name — Address of the table name to search for; names are case sensitive.

buffer — Address of buffer to store information collected.

bufsiz — Size of buffer in bytes.

itemsiz — The number of bytes to store from each table. If itemsiz is less than the table size, only as many as complete fields from each table found are written in the buffer. If itemsiz is greater than the table size, the excess area is not modified.

key — Key from which to continue searching. The key value depends on the table type. Each table allowing LOOKUP specifies a key for continued search. The LOOKUP SVC continues the search from the first item after the key. A key value of 0 always starts the LOOKUP search from the beginning of the table.

Return Code:

nfound — Number of tables found. LOOKUP stops searching when the end of the buffer is reached or there are no more tables. If the last table does not fit into the remaining buffer space, it is discarded.

ret — Error Code

bit 4: 1 = Shared
0 = Exclusive

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

bit 8: 1 = Force logical remount
0 = Do not force logical remount

bit 9: 1 = Force physical remount
0 = Do not force physical remount

bits 10 - 12 are 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 is reserved (must be 0)

name Address of file, pipe, or device name

Return Code:

fnum file number
ret Error Code

Description: The OPEN SVC opens an existing file and returns a 32-bit file number used for subsequent I/O. "File" in this context refers to disk files, pipes, and device files used to communicate with printers, mouses, consoles, and special devices. FlexOS sets the file pointer to 0 when you open the file.

Use flag bits 0 through 3 to request the file access privileges--read, write, execute, and delete/set. Use flags 4, 5, and 6 to set the access mode--shared versus exclusive, shared read only versus shared read/write when shared, and shared versus unique file pointer. The use of these flags to monitor file access differs slightly from one type of file to another. See the sections in this manual on disk file, console, pipe, and special device management for the description of flag use with these types of files.

Set flag bit 6 when you want two or more processes to share the same file pointer; this feature is only available to processes with the same family identification number (FID). Each process sharing the pointer must have this flag set. When this bit is set, the value of flag bit 1 is assumed to be 1; the actual value is ignored.

Set bit 7 to accept reduced access privileges. The file's governing privileges for owner, group, and world categories are set when it is created. Reduced access is an issue when a disk label's security flag bit is set and you request a privilege level not available to a process with your ID and group number. Set this flag to 1 if you can accept reduced access; FlexOS ANDs the file's R, W, E, and D privileges corresponding to your category with those you requested to determine the privileges you actually get. Set this flag to 0 if you cannot accept reduced access; FlexOS returns an error code when the privileges do not match.

7.31 RETURN

C Interface:

```
LONG        emask;

ret = s_return(emask);

ret = __osif(F_RETURN,emask);
```

Parameters:

emask	Event mask of completed event

Return Code:

ret	return code of asynchronous SVC

Description: The RETURN SVC retrieves the return code of an asynchronous SVC. If the event is not complete, FlexOS waits for it to complete before returning from the RETURN call. Use WAIT or STATUS to determine if the event has completed. The return code is the code that would have been returned if the SVC had not been called synchronously. Once the RETURN SVC has been called, the event's emask bit is cleared.

Note: You cannot use RETURN for events with a software interrupt (swi). The event's completion is provided to the swi and is not kept available to the parent process.

7.32 RWAIT

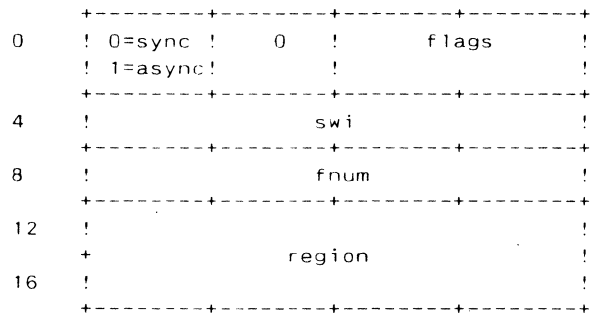
C Interface:

```
RECT        *region;

position = s_rwait(flags,fnum,region);
emask = e_rwait(swi,flags,fnum,region);

ret = __osif(F_RWAIT,&parmblk);

parmblk:
```



Parameters:

flags

bit 0: 0 = return on entry from rectangle
1 = return on exit to rectangle

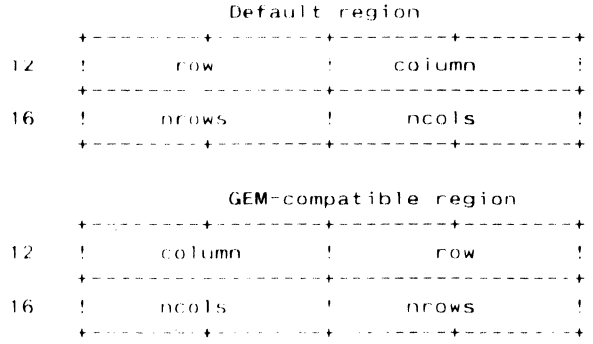
bit 1: 0 = no clip
1 = clip to visible view of the window

bit 2: 0 = default region definition
1 = GEM-compatible region definition

bits 3-15 are reserved and must be 0.

fnum File number of open mouse file

region RECT structure describing a rectangular area of the screen associated with the mouse.



Return Code:

ret Error Code

Position (see diagram below)

Description: The RWAIT SVC allows a process to detect the mouse entering or exiting a described region of the screen.

If bit 0 = 0, RWAIT checks only inside the current view on the screen. If bit 0 =1, the rectangle can be outside the visible view to complete the event.

7.33 SEEK

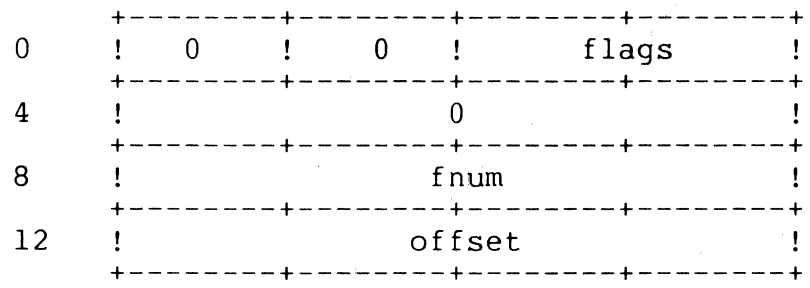
C Interface:

```
LONG        fnum,offset;
UWORD       flags;

position = s_seek(flags,fnum,offset);

ret = __osif(F_SEEK,&parmblk);

parmblk:
```



7.39 WAIT

C Interface:

```
LONG        events,cmask;

cmask = s_wait(events);

ret = __osif(F_WAIT,events);
```

Parameters:

events	Logical OR of emasks to wait for

Return Code:

cmask	Bit map of completed events
Error Code	e_emask if all the events are invalid

Description: The WAIT SVC causes the calling process to wait for an asynchronous event to occur. Specify one or more events by their emask in the WAIT events argument. FlexOS returns when one of these events has run to completion. For events that do not have a software interrupt, the cmask return code indicates which event completed. Subsequently, call the RETURN SVC to retrieve the return code of the completed event. This also releases that emask so it can be reused.

You can wait on events that have a software interrupt (swi). However, the event bit in the cmask returned is 0 rather than 1 when WAIT returns. Also, do not call RETURN to retrieve the completion code after WAIT returns--the completion is no longer available having already been provided to the swi for handling.

If multiple events are being waited on and one or more are invalid they are ignored. If they are all invalid, an error E_EMASK is returned.

7.40 WRITE

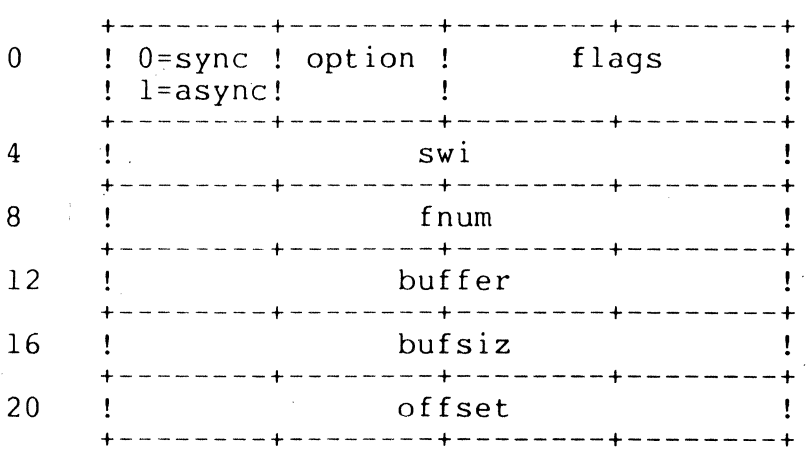
C Interface:

```
LONG        fnum,bufsiz,offset,nbytes;
BYTE        option,*buffer;
UWORD       flags;

nbytes = s_write(flags,fnum,buffer,bufsiz,offset);
emask = e_write(swi,flags,fnum,buffer,bufsiz,offset);

ret = __osif(F_WRITE,&parmblk);
```

parmblk:



System Tables

System status and parameter values are available to applications through the GET, SET, and LOOKUP SVCs which operate on a set of formalized data structures that comprise FlexOS's system tables. This section presents descriptions of the system tables in alphabetical order.

The GET SVC transfers the table to a buffer in the application's memory space. The SET SVC changes values in a table. For both SVCs, the table is identified by its number and, when that table type has more than one version, a unique ID number. The LOOKUP SVC searches for and retrieves tables of the same type. Each table that can be accessed with LOOKUP has a key value field; use this field to specify a starting point for the search.

The GET, SET, and LOOKUP SVCs will not access all of the system tables. Table 8-1 lists each of the system tables and the SVCs used to access them. Also listed in Table 8-1 are each table's number, ID, and key value.

Table 8-1. System Table Access

Table No. & Name		GET	SET	Unique ID	LOOK UP	Key	Description
0H	PROCESS	X	X	pid	X	pid	Process information
1H	ENVIRON	X	X	0			Process environment
2H	TIMEDATE	X	X	0			System time of day
3H	MEMORY	X		0			System memory use
10H	PIPE	X		fnum	X	key	Pipe information
20H	DISKFILE	X	X	fnum	X	key	Disk file information
21H	DISK	X	X	fnum			Disk device information
30H	CONSOLE	X	X	fnum			Console file information
31H	PCONSOLE	X	X	fnum			Console device information
32H	VCONSOLE	X	X	fnum	X	VCNUM	Console information
33H	MOUSE	X	X	fnum	X		Mouse information
40H	SYSTEM	X	X	0			Global system information
41H	FILNUM	X		fnum	X	fnum	File number's table
42H	SYSDEF				X	key	System logical name table
43H	PROCDEF				X	key	Process logical name table
44H	CMDENV	X		pid			Command environment
45H	DEVICE				X	key	Device information
46H	PATHNAME				X	none	Full path name
71H	PRINTER	X	X	fnum			Printer device information
75H	MOUSE DT	X	X	fnum			Mouse driver table information
81H	PORT	X	X	fnum			Port device information
82H+	SPECIAL	X	X	fnum			Special device information

In the following system table descriptions, only those fields marked **R/W** are read-write; all other fields are read-only. In all bit-mapped values the bits for which there are no options are reserved and must be 0.

Note: FlexOS does not maintain memory representations for the tables described in this section. The corresponding resource manager or driver constructs them only when you call the GET, SET, or LOOKUP SVCs.

- **KMODE (R/W)**: Keyboard mode

  bit 0: 1 = Disable Control-C
  0 = Control-C attempts external abort

  bit 1: 1 = Disable Control-S/Control-Q
  0 = Allow Control-S/Control-Q

  bit 2: 1 = Disable keyboard s_xlat translation table
  0 = Translate keys

  bit 3: 1 = Disable ESC sequence decoding
  0 = Support ESC sequence

  bit 4: 1 = Characters are 16-bit values
  0 = Characters are 8-bit values

  bit 5: 1 = Disable echo
  0 = Echo input characters on screen

  bit 6: 1 = Disable CTRL-Z
  0 = CTRL-Z = end of file

  bit 7: 1 = Enable toggle characters
  0 = Disable toggle characters

  bit 8: 1 = Convert <LF> or <CR> to <CR><LF>
  0 = Do not convert <LF> or <CR>

  bit 9: 1 = Do not echo carriage returns
  0 = Echo carriage returns

  bit 10: 1 = Do not echo <CR> on any delimiter
  0 = Echo <CR> on any delimiter

- **CURROW (R/W)**: Current cursor row position
- **CURCOL (R/W)**: Current cursor column position

- **NROWS:** Height of virtual screen in character rows
- **NCOLS:** Width of virtual screen in character columns
- **VCNUM:** Decimal number of virtual console
- **TYPE:** Type of virtual console

  bit 0: 1 = Graphics capability
  0 = Character only

  bit 1: 1 = No numeric keypad
  0 = Keypad

  bit 2: 1 = Mouse support
  0 = No mouse support

  bit 3: 1 = Color
  0 = Black and white

  bit 4: 1 = Memory-mapped video
  0 = Serial device

  bit 5: 1 = Currently in graphics mode
  0 = Currently in character mode

- **CNAME:** Physical console device name

Each console file opened has a corresponding CONSOLE table. The TAHEAD, CURROW, and CURCOL values are initialized to 0 when the console file is opened. NROWS and NCOLS correspond to the rows and columns set in the virtual console. SMODE and KMODE are initialized to 0; TYPE and CNAME are inherited from the parent console.

GET and SET the CONSOLE table using as the ID the file number returned when you OPENed the file vcxxx/console. Do not use the file number returned when you CREATEd the virtual console. For most applications, this file number is contained in the stdout--the screen file number--and stdin--the keyboard file number--in the ENVIRON table. Stdin and stdout can have the same or different file numbers.

Use SET to change the cursor position and the screen and keyboard modes.

- **INSTAT:** Installation status

  0x00 - Not installed
  0x01 - Requires subdriver
  0x02 - Owned by the Miscellaneous Resource Manager
  0x03 - Owned by another driver
  0x04 - Optional subdriver

- **OWNERID:** Significant 16 bits of the key field of the owner's DEVICE table entry. Use this value with a LOOKUP to find the driver that owns this subdriver. This field is only valid when INSTAT has a value of 0x03.

The DEVNAME, TYPE, ACCESS, and KEY values are established when the device is installed and do not change. The ACCESS flags override conflicting requests made by programs when they open the device.

The INSTAT and OWNERID values are also static except for subdrivers assigned to different drivers. In this case, the current values are subject to change as the driver is linked and unlinked to different owners.

You must use the LOOKUP SVC to get DEVICE tables. Wildcards can be used in the LOOKUP device name specification.

8.4 DISK Table

Number	GET?	SET?	LOOKUP?
21H	Yes	Yes	No

ID: File number returned by OPEN
Key: none

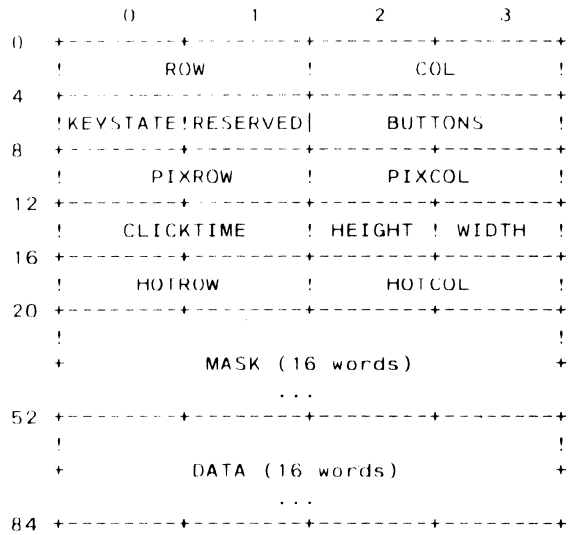
The DISK table describes a disk driver. All fields are read-only except the label options.

8.9 MOUSE Table

Number	GET?	SET?	LOOKUP?
33H	Yes	Yes	No

ID: File number returned by OPEN
Key: none

The MOUSE table describes a pointing device. Every installed pointing device has a MOUSE table. The initial values are set by the driver and you can set all of them except for the PIXROW and PIXCOL.



- **ROW (R/W):** Current row position of mouse
- **COL (R/W):** Current column position of mouse

- **KEYSTATE:** Keyboard state of the right Shift, left Shift, Control, and Alt keys

  Bit 0 right Shift key
  Bit 1 left Shift key
  Bit 2 Control key
  Bit 3 Alt key

  0 - up position
  1 - down position

- **BUTTONS:** The least significant bit is the **leftmost** button. Total buttons supported is 16.
- **PIXROW:** Number of mickeys per pixel for rows
- **PIXCOL:** Number of mickeys per pixel for columns
- **CLICKTIME (R/W):** Click interval in milliseconds (usually 174)
- **HEIGHT (R/W):** Height of mouse form
- **WIDTH (R/W):** Width of mouse form
- **HOTROW (R/W):** Hot row of mouse form
- **HOTCOL (R/W):** Hot column of mouse form
- **MASK (R/W):** On a bit map screen, a 16 x 16 pixel rectangle that masks the effect of the DATA rectangle.
- **DATA (R/W):** On a bit map screen, a 16 x 16 pixel rectangle to "BLT" to the screen given the mask.

The ROW and COL values are updated by the Console Resource Manager to indicate the current mouse location. You can, however, set these values to move the mouse form to a location without device input. The HEIGHT and WIDTH values have a maximum value of 4, but can be less. If either is less, the length of the MASK and DATA fields is not affected.

8.9a MOUSE DRIVER Table

Number	GET?	SET?	LOOKUP?
75H	Yes	Yes	No

ID: File number returned by OPEN
Key: none

The MOUSE Driver is a device driver for a pointing device such as a mouse, tablet, touch screen, light pen, joystick, or other similar device. Every installed pointing device has a MOUSE Driver table. Normally, a MOUSE Driver is optionally owned by the console driver as a subdriver. The MOUSE driver itself requires a PORT subdriver to connect to the physical I/O port.

After loading and linking a PORT Driver, the MOUSE Driver is owned by the Resource Manager and then you can use GET and SET on this table. However, after the MOUSE Driver is linked to and optionally owned by a CONSOLE Driver, this table is inaccessible.

The CONSOLE Driver sets the ROW_MAX and COL_MAX values for maximum resolution in graphics mode (e.g. 640 x 200) using a SET call, then makes a GET call to read the values for BUTTONS, PIXROW, and PIXCOL from the MOUSE Driver.

Note: All row and column values are absolute pixel values of the physical screen.

Sample installation script:

```
dvrload mouse0:  hd0:\drivers\mdrv.drv lnrws ;load the mouse driver
dvrlink mouse0: pt0:                        ;link it to serial port 0
dvrlink con0: mouse0:                       ;make mouse driver a subdriver
                                            ;of the console driver

if you want to connect it to serial port 1:

dvrunlk mouse0:                             ;unlink driver from console
dvrload mouse0:  hd0:\drivers\mdrv.drv lnrws ;load the mouse driver again
dvrlink mouse0: pt1:                        ;link it to serial port 1
dvrlink con0: mouse0:                       ;make mouse driver a subdriver
                                            ;of the console driver
```

0	ROW_MAX	COL_MAX
4	RESERVED	NUM_BUTTONS
8	PIXROW	PIXCOL
10	DOUBLE_Y	DOUBLE_X
16	CURRENT_ROW	CURRENT_COL
20	BUTTON_STATE	RESERVED

- **ROW_MAX (R/W)**: Maximum value of row position of MOUSE in pixels
- **COL_MAX (R/W)**: Maximum value of column position of MOUSE in pixels
- **NUM_BUTTONS**: Number of buttons supported by driver
- **PIXROW**: Number of internal pixels for one row (set by driver)
- **PIXCOL**: Number of internal pixels for one column (set by driver)
- **DOUBLE_Y (R/W)**: Double the internal delta_y values when bigger than this value. Optimized for quicker mouse movement
- **DOUBLE_X (R/W)**: Double the internal delta_x values when bigger than this value. Optimized for quicker mouse movement
- **CURRENT_ROW**: Current row position; valid values are 0 to ROW_MAX
- **CURRENT_COL**: Current column position; valid values are 0 to COL_MAX
- **BUTTON_STATE**: Current button state. Each bit represents a mouse button, with the least significant bit representing the **leftmost** button.

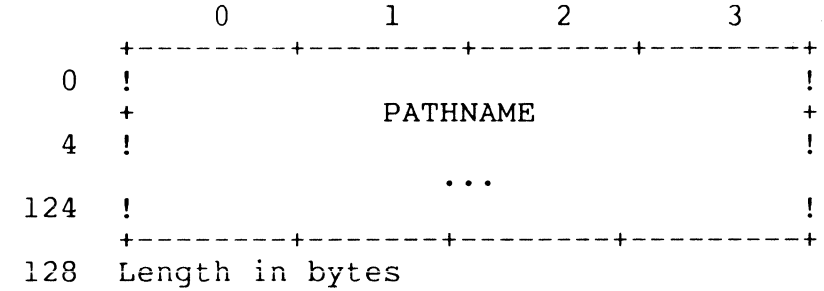
  bit = 0 button not pressed
  bit = 1 button pressed

8.10 PATHNAME Table

Number	GET?	SET?	LOOKUP?
46H	No	No	Yes

ID: none
Key: none

The PATHNAME table contains the fully-expanded path name for a defined symbol. LOOKUP is the only way to retrieve a PATHNAME table; you cannot SET or GET a PATHNAME.



The PATHNAME table consists of a single 128 byte field. Only one path is ever returned when you lookup a defined symbol. If the symbol specified starts with a defined name, the prefix is substituted for the symbol. If the first name in the prefix is itself a defined symbol, the substitution is made again. The search and substitute routine is repeated until no prefix is found for the starting name.

The SYSDEF and PROCDEF tables are searched when you lookup the PATHNAME table. (DEFINE only looks in one or the other.) These tables are searched for the first name in the specification only.

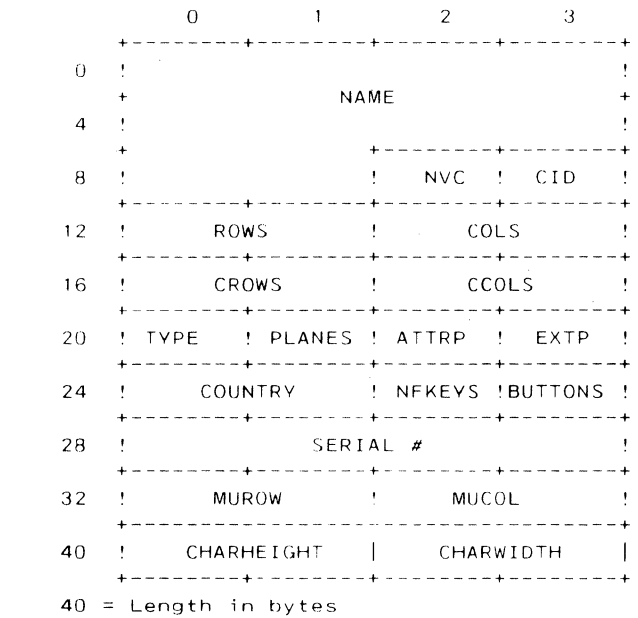
Wildcard characters can be used but they are not expanded; for example, as asterisk is interpreted only as an asterisk.

8.11 PCONSOLE Table

Number	GET?	SET?	LOOKUP?
31H	Yes	Yes	No

ID: File number returned by OPEN
Key: none

The PCONSOLE table describes a physical console device. Each console installed has its own PCONSOLE table. All parameters are read-only except the country code.



- **NAME**: Console device name
- **NVC**: Current number of virtual consoles
- **CID**: Physical console ID number

- **ROWS:** On graphic console devices, this is the number of rows of pixels. On character console devices, this is the number of character rows and is the same as CROWS.
- **COLS:** On graphic console devices, this is the number of pixels in a row. On character console devices, this is the number of character columns and is the same as CCOLS.
- **CROWS:** The number of rows of characters
- **CCOLS:** The number of columns of characters
- **TYPE:** Type of console

  bit 0: 1 = Graphics capability
  0 = Character only

  bit 1: 1 = No numeric keypad
  0 = Keypad

  bit 2: 1 = Mouse supported
  0 = No mouse supported

  bit 3: 1 = Color
  0 = Black and white

  bit 4: 1 = Memory-mapped video
  0 = Serial device

  bit 5: 1 = Currently in graphics mode
  0 = Currently in character mode

- **PLANES:** Planes supported

  Bit 0: 1 = Character plane supported
  0 = No character plane

  Bit 1: 1 = Attribute plane supported
  0 = No attribute plane

  Bit 2: 1 = Extension plane supported
  0 = No extension plane

- **ATTRP:** Bit map of attribute plane bits supported
- **EXTP:** Bit map of extension plane bits supported
- **COUNTRY (R/W):** Country code; in applications that support multiple character sets, use this value to select a specific set. Appendix C lists the country codes.
- **NFKEYS:** Number of function keys supported
- **BUTTONS:** Number of mouse buttons supported
- **SERIAL #:** Mouse serial number
- **MUROW:** Mouse sensitivity in mickey units per row
- **MUCOL:** Mouse sensitivity in mickey units per column
- **CHARHEIGHT:** Height of character cell in pixels
- **CHARWIDTH:** Width of character cell in pixels

The PCONSOLE values are set by the driver. The Console Resource Manager updates the NVC value as you create and delete virtual consoles on this console.

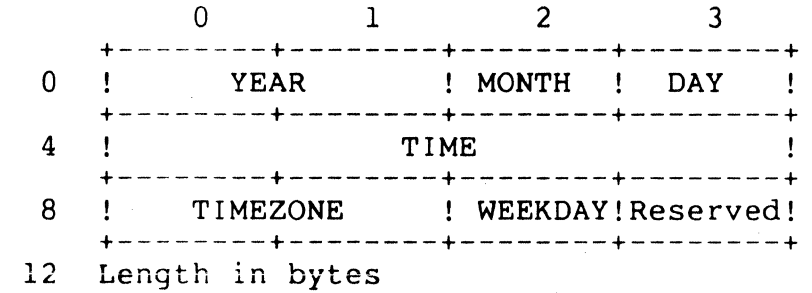
To GET and SET a PCONSOLE table (LOOKUP cannot be used), OPEN the device and use the file number returned as the GET and SET ID number. In your OPEN call, the only access mode flag bit you can set is bit 0 and you only need set it if you want to change the country code.

8.20 TIMEDATE Table

Number	GET?	SET?	LOOKUP?
2H	Yes	Yes	No

ID: 0
Key: none

The TIMEDATE table contains the system time of day. All fields are read/write except WEEKDAY. The time is maintained by the kernel once the starting is set. Use SET to establish the starting time.



- **YEAR (R/W)**: Year; a literal value (for example, 1987 = 1987)
- **MONTH (R/W)**: Month; 1 - 12
- **DAY (R/W)**: Day of the month; 1 - 31
- **TIME (R/W)**: Number of milliseconds since midnight
- **TIMEZONE (R/W)**: Minutes from Universal Coordinated Time
- **WEEKDAY**: Day of the week; 0 = Sunday, 6 = Saturday

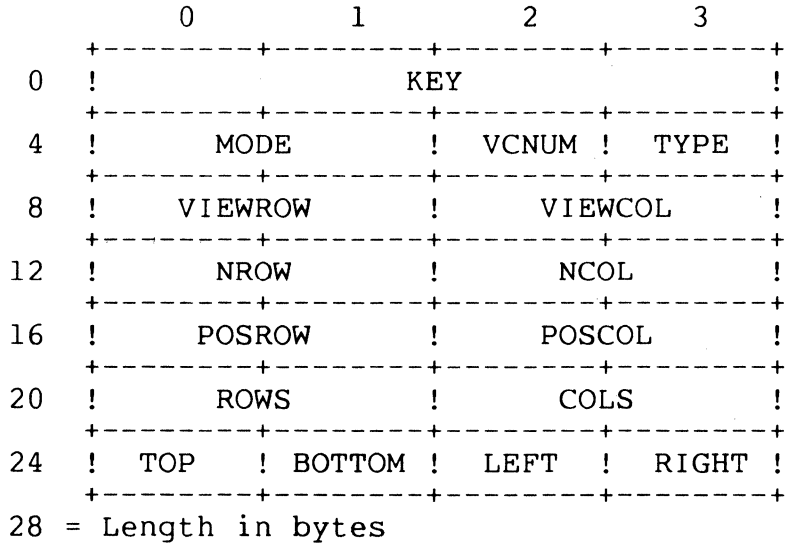
You use an ID value of 0 to GET and SET the TIMEDATE table.

8.21 VCONSOLE Table

Number	GET?	SET?	LOOKUP?
32H	Yes	Yes	Yes

ID: File number returned by CREATE
Key: VCNUM assigned when virtual console created

The VCONSOLE table describes a virtual console. Table values are established when you CREATE the console. Use read/write fields to modify window size, location on the virtual console, and placement on the parent console.



- **KEY:** Key field for LOOKUP

- **MODE (R/W):** Window mode

  bit 0: 1 = Freeze borders
  0 = Synchronize borders (See Note 1, below)

  bit 1: 1 = Allow auto view change (See Note 2, below)
  0 = Keep view fixed

  bit 2: 1 = Keep cursor on edge on auto view change
  0 = Center cursor on auto view change

  bit 3: 1 = Auto view change on output
  0 = Auto view change on input

- **VCNUM:** Decimal virtual console number

- **TYPE:** Type of console.

  bit 0: 1 = Graphics capability
  0 = Character only

  bit 1: 1 = No numeric keypad
  0 = Keypad

  bit 2: 1 = Mouse supported
  0 = No mouse supported

  bit 3: 1 = Color
  0 = Black and white

  bit 4: 1 = Memory-mapped video
  0 = Serial device

  bit 5: 1 = Currently in graphics mode
  0 = Currently in character mode

- **VIEWROW (R/W):** Row coordinate on the virtual console view upper lefthand corner

- **VIEWCOL (R/W):** Column coordinate on the virtual console view upper lefthand corner

- **NROW (R/W):** Number of character rows of the view
- **NCOL (R/W):** Number of character columns of the view
- **POSROW (R/W):** Row coordinate on parent virtual console of view upper lefthand corner
- **POSCOL (R/W):** Column coordinate on parent virtual console of view upper lefthand corner
- **ROWS:** Number of character rows in the virtual console
- **COLS:** Number of character columns in the virtual console
- **TOP:** Height in character rows of the top border
- **BOTTOM:** Height in character rows of the bottom border
- **LEFT:** Width in character columns of the left border
- **RIGHT:** Width in character columns of the right border

Notes:

1. Use bit 0 to freeze a border so that intermediate states are not displayed when you make changes to the border file contents. Before you change the border file contents, set this bit. After you have completed the changes, reset the bit. Normally, keep this flag at 0 so that the borders change as you make changes to the window dimensions and location.
2. Bits 1 through 3 determine whether the window view changes to keep the cursor on-screen or the view remains fixed on the same virtual console coordinates regardless of cursor location. If the cursor leaves the window and bit 2 = 1, bit 3 determines whether the view changes when the cursor leaves the view (output) or when the application READs the keyboard.

Table B-2. Low-order Word Error Code Ranges

Error Code Range	Source
0000 - 3FFF	Drivers
4000 - 407F	Errors Common to All Resource Managers
4080 - 40FF	Supervisor
4100 - 417F	Memory
4180 - 41FF	Kernel
4200 - 427F	Pipe and Miscellaneous Resource Managers
4280 - 42FF	Console System
4300 - 437F	File System
4400 - FFFF	Reserved

For the source of one of the common error codes, see the low byte in the high order word. The remaining tables in this appendix list define the error messages by their source. No error codes are currently associated with the Pipe, Console and Miscellaneous Resource Managers.

Table B-3. Driver Error Codes

Mnemonic	Code	Description
E_WPROT	0x00	Write protect violation
E_UNITNO	0x01	Illegal unit number
E_READY	0x02	Drive not ready
E_INVCMD	0x03	Invalid command issued
E_CRC	0x04	CRC error on I/O
E_BADPB	0x05	Bad parameter block
E_SEEK	0x06	Seek operation failed
E_UNKNOWNMEDIA	0x07	Unknown media present
E_SEC_NOTFOUND	0x08	Requested sector not found
E_DKATTACH	0x09	Attachment did not respond
E_WRITEFAULT	0x0A	Write fault
E_READFAULT	0x0B	Read fault
E_GENERAL	0x0C	General failure
E_MISSADDR	0x0D	Missing address mark
E_NEWMEDIA	0x0E	New media detected
E_DOOROPEN	0x0F	Door has been opened

Table B-4. Error Codes Shared by Resource Managers

Mnemonic	Code	Description
E_SUCCESS	0x0L	No Error
E_ACCESS	0x4001	Cannot access file--ownership differences
E_CANCEL	0x4002	Event Cancelled
E_EOF	0x4003	End of File
E_EXISTS	0x4004	File (CREATE) or device (INSTALL) exists
E_DEVICE	0x4005	Device does not match or not found; for RENAME, on different devices
E_DEVLOCK	0x4006	Device is LOCKed
E_FILENUM	0x4007	Bad File Number
E_FUNCNUM	0x4008	Bad function number
E_IMPLEMENT	0x4009	Function not implemented
E_INFOTYPE	0x400A	Illegal Infotype for this file
E_INIT	0x400B	Error on driver initialization
E_CONFLICT	0x400C	Cannot access file due to current usage; for DELETE on open file or directory with files; for INSTALL, attempted replacement of driver in use
E_MEMORY	0x400D	Not enough memory available
E_MISMATCH	0x400E	Function mismatch--file does not support attempted function; for INSTALL, mismatch of subdrive type
E_NAME	0x400F	Illegal file name specified
E_NO_FILE	0x4010	File not found; for CREATE, device or directory does not exist
E_PARM	0x4011	Illegal parameter specified; for EXCEPTION, an illegal number
E_RECSIZE	0x4012	Record Size does not match request
E_SUBDEV	0x4013	INSTALL only: Sub-drive required
E_FLAG	0x4014	Bad Flag Number
E_NOMEM	0x4015	Non-existent memory

Table B-4. (Continued)

Mnemonic	Code	Description
E_MBOUND	0x4016	Memory bound error
E_ILLINS	0x4017	Illegal instruction
E_EDIVZERO	0x4018	Divide by zero
E_EBOUND	0x4019	Bound exception
E_OFLOW	0x401A	Overflow exception
E_PRIV	0x401B	Privilege violation
E_TRACE	0x401C	Trace
E_BRKPT	0x401D	Breakpoint
E_FLOAT	0x401E	Floating point exception
E_STACK	0x401F	Stack fault
E_NOTON286	0x4020	General Exception
E_EMI	0x4021	Emulated instruction group 1

Table B-5. Supervisor and Memory Error Codes

Mnemonic	Code	Description
E_ASYNC	0x4080	Function does not allow asynchronous I/O
E_LOAD	0x4082	Bad load format
E_LOOP	0x4083	Infinite recursion (99 times) on prefix substitution; for INSTALL, subdrive type mismatch
E_FULL	0x4084	File number table full
E_DEFINE	0x4085	DEFINE only: illegal, or undefined name
E_UNIT	0x4086	Too many driver units
E_UNWANTED	0x4087	Driver does not need subdriver
E_DVRTYPE	0x4088	Driver returns bad driver type
E_LSTACK	0x4089	Stack not defined in load header
Memory Error Codes		
E_POOL	0x4100	Out of memory pool
E_BADADD	0x4101	Specified bad address

Table B-6. Kernel Error Codes

Mnemonic	Code	Description
E_OVERRUN	0x4180	Flag already set
E_FORCE	0x4181	Return code of aborted process
E_PDNAME	0x4182	Process ID not found on abort
E_PROCINFO	0x4183	COMMAND only: no procinfo specified
E_LOADTYPE	0x4184	COMMAND only: invalid loadtype
E_ADDRESS	0x4185	CONTROL only: invalid memory access
E_EMASK	0x4186	Invalid event mask, or out of events
E_COMPLETE	0x4187	Event has not completed
E_STRL	0x4188	Required SRTL could not be found
E_ABORT	0x4189	Process cannot be terminated
E_CTRLC	0x418A	Process aborted by Ctrl-C
E_GO	0x418B	Slave process running
E_INSWI	0x418C	Not in SWI context; not a swi process
E_UNDERRUN	0x418D	Flag already pending

Table B-7. Disk Error Codes

Mnemonic	Code	Description
E_SPACE	0x4300	Insufficient space on disk or in directory
E_MEDIACHANGE	0x4301	Media change occured
E_MEDCHGERR	0x4302	Detected media change after a write
E_PATH	0x4303	Bad path
E_DEVCONFLICT	0x4304	Devices locked exclusively
E_RANGE	0x4305	Address out of range
E_READONLY	0x4306	RENAME or DELETE on R/O file
E_DIRNOTEMPTY	0x4307	DELETE of non-empty directory
E_BADOFFSET	0x4308	Bad offset in read, write or seek
E_CORRUPT	0x4309	Corrupted FAT
E_PENDLK	0x430A	Cannot unlock a pending lock
E_RAWMEDIA	0x430B	Not FlexOS media
E_FILECLOSED	0x430C	File closed before asynchronous lock could be completed
E_LOCK	0x430D	Lock access conflict
E_FATERR	0x430E	Error while reading FAT

Utility return codes follow the same format of operating system error return codes, as illustrated in Figure B-1, with the following exceptions:

- Utility return codes are positive numbers (LONGS) because the high order bit (31) is always zero.
- When possible, you should use the error codes listed in Table B-8 in the error code field (bits 0-15).
- You can designate given modules within an application in the source field (bits 16-23).

To return errors generated within your application, OR the source field (module) with the error code field. For example, to indicate that an application has detected a parameter error, use:

```
return( UR_SOURCE | UR_PARM );
```

Do not OR a source field value with UR_SUCCESS, which is a LONG of zeroes.

Table B-8. Utility Return Codes

Mnemonic	Code	Description
UR_SOURCE	(LONG)0	Utility return
UR_SUCCESS	(LONG)0	Success
URPARM	0x0001	Parameter error
UR_CONFLICT	0x0002	Contention conflict
UR_UTERM	0x0003	Terminated by user
UR_FORMAT	0x0004	Data structure format error
INTERNAL	0x0005	Internal utility error
UR_UR_DOSERR	0x0006	PC DOS error

End of Appendix B

VDI Demo Programs

Diskette **PTK 3** contains three programs that demonstrate the capabilities of the Virtual Device Interface (VDI):

thing [-I <# of points>]

This program draws a simple line-figure; it is useful for testing lines and line colors. The **-I** switch tells **thing** to draw a design with the specified number of points around the perimeter. By default, **thing** draws the figures with from four to ten points around the perimeter.

polyline

This program is documented in the GEM VDI reference manual.

logos

This program draws a number of Digital Research logos on the screen, using the copy transparent function, draws DRI's slogan over that using a one of the fonts loaded, and then cycles the colors on 16 color screens (on an EGA device).

All of these programs have the following switches in common:

-f device Sends output to the specified device instead of the screen.

-m Emulate monochrome on color devices.

-c colors number Set the maximum number of colors to the argument. If you specify too many colors, the number defaults to the maximum available. **-m** overrides this switch.

End of Section 12