CHAPTER 12: DEVICE DRIVERS

Introduction

Device drivers are the interface between MOS and peripheral devices, such as disk drives, monitors and terminals. They are also used for installing certain system features. This capability provides a great deal of flexibility to MOS. This chapter discusses what a device driver should look like, how to create one, and how the drivers are installed.

A device driver contains a special header in front of it which MOS uses to communicate with the driver. This header provides pointers to the next driver in the chain and to the driver's primary and secondary entry points. It also provides device attributes and gives the name of the device or the number of units the device controls.

Once MOS has loaded itself into memory, it opens the CONFIG.SYS file and begins reading the information from it. External device drivers are loaded into the SMP when MOS processes DEVICE= statements in the CONFIG.SYS file. Once the driver is loaded, MOS prepares the command line string, if any, by removing any tilde (~) characters that MOS uses as a line extender and the associated carriage return/line feed. This way, when the driver parses the line, it knows a carriage return/line feed indicates the end of the string. MOS then calls the driver to initialize it.

MOS allows drivers to be dynamically added and removed through the use of the ADDDEV.COM utility. One important difference between using ADDDEV and having a driver loaded through CONFIG.SYS is the way that interrupt vectors are handled. This will be covered in more detail later in this chapter.

There are two basic types of device drivers, Character and Block. Character devices handle I/O as sequential groups of single characters. Character devices have unique names, such as COM1 and LPT1. Because of this, character drivers can support only one device. The device can be accessed by name, just like files, via the handle calls to MOS, i.e. open, read, write, close. See the chapter on System Calls for further information.

Block devices perform non-sequential I/O in pieces called blocks. Block devices are usually disks or diskettes. The block size is usually the same as the physical sector size of the media. If the physical disk has a sector size of 512 bytes, then the block size is 512 bytes.

Block devices can control multiple logical units. For instance, MOS's diskette device driver controls two logical units, A and B. The fixed disk driver controls up to eight logical units. Because block devices can control multiple units, they don't have a name and can't be accessed via any of the file related MOS function calls as character devices can.

Block devices are referred to using identifier letters A, B, C, and so on. Block drivers refer to their sub-units using 0-based sub-unit codes.

MOS keeps track of the number of logical units installed and passes a beginning drive ID number to each driver. For example, if the first block driver controls 2 units, it has drives A and B. If the next driver loaded can control eight units, it would have drives C, D, E, F, G, H, I, J. Should another driver be loaded with only one unit, it would be K.

Follow these guidelines when creating device drivers:

1. It should originate at offset zero within its code segment.

2. If you load multiple drivers in one file, be sure you link the pointer fields in the device headers to maintain the driver chain. Set the last one in the chain to -1 (0FFFFh). All drivers within the file must have the same code segment. To link the drivers, put the offset into the first word of the link field in the driver header. The segment address will be filled in by MOS when it loads the file. The link address is stored with the offset portion preceding the segment portion.

3. The driver must tell MOS where it ends in memory. This entails returning a double word pointer in the request header that points to the first byte beyond the end of the driver. If the file contains multiple drivers, all drivers should return the same point as the end. This allows the system to maximize memory use by allowing the driver to throw away its INIT code.

4. The driver must put the addresses of its primary and secondary entry routines in the device header. It must set its name or unit number in the name/unit field.

5. The driver must set its attribute field.

The Device Header

The following pages give a detailed description of the device header and its associated fields. The header is as follows:

Field	Length
Pointer to next driver	4 bytes (double word)
Attribute	2 bytes (word)
Pointer to primary entry	2 bytes (word)
Pointer to secondary entry	2 bytes (word)
Name/Unit	8 bytes (bytes)

A sample header, in assembler language, might be:

```
dd      -1              ; driver chain pointer
dw      8000h           ; this is a character device
dw      strategy        ; pointer to primary entry
dw      interrupt       ; pointer to secondary entry
db      'SAMPLE  '      ; device name - 8 chars
```

THE LINK FIELD

The link field is a double word pointer which MOS uses to link the driver into the chain. The values stored there are the offset and segment addresses of the next driver. If the value is -1, it's the end of the chain. MOS enters these values into the driver. In the example it is already -1 because the driver is last in the chain.

THE ATTRIBUTE FIELD

This table will give a brief description of the bits of the attribute word in the device header. An asterisk in the columns Char or Block indicates which type of device the description is pertinent to. More details will follow.

Bit	Description	Char	Block
15	Device type. 0=block, 1=character		
14	IOCTL support. 0=No, 1=Yes	*	*
13	Media Format. 0=Media byte. 1=BPB		*
13	Output-until-busy, 0=No, 1=Yes	*	
12	Reserved. MUST be zero!	*	*
11	Removable Media Bit. 0=No, 1=Yes		*
11	Open/Close calls. 0=No, 1=Yes	*	
10	Reserved. MUST be zero!	*	*
9	Reserved. MUST be zero!	*	*
8	Reserved. MUST be zero!	*	*
7	Reserved. MUST be zero!	*	*
6	Get/Set Logical Dev. 0=No, 1=Yes		*
5	Reserved. MUST be zero!	*	*
4	Reserved. MUST be zero!	*	*
3	Current Clock Device. 0=No, 1=Yes	*	
2	Current NUL Device. 0=No, 1=Yes	*	
1	Current STDOUT Device. 0=No, 1=Yes	*	
0	Current STDIN Device. 0=No, 1=Yes	*	
0	Generic IOCTL. 0=No, 1=Yes		*

Bit 15: This bit indicates to the system which type of device it is dealing with. The bit is 1 if it is a character device, 0 for a block device.

Bit 14: This bit tells the system that the device can use control strings. Both types of drivers may use this bit. Control strings are passed directly to the driver via the IOCTL function call (44h). Normally, information is passed to the driver through read/write calls. IOCTL can be used to pass special data or information to the driver. This data can be used by the driver to control different aspects of the device (such as setting character print size or changing the mode of a monitor). If the device cannot use control strings, it should set this bit to zero so that MOS can return the appropriate error code if an application makes an IOCTL call to the device.

Bit 13: Character device drivers use this bit to tell the system whether or not the driver supports output-until-busy.

Output-until-busy is a protocol which allows the driver to send characters to the device until the device can no longer accept them, i.e. its input buffer is full. Once the device has processed its received characters, it signals the driver to send more. The flow control mechanism is established between the driver and the device. The driver could use XON/XOFF, DTR/DSR, RTS/CTS or whatever has been designed into the device.

MOS is only concerned with whether or not the driver is using some type of protocol. Bit 13 indicates whether or not it is. If the device is not ready, the driver must return an error 2, which is "device not ready".

For block devices, this bit tells MOS how the driver determines the media type in a drive. If this bit is zero, the driver uses the media descriptor byte. If this bit is one, the driver uses the BPB to determine the media type.

Bit 12: This bit is reserved.

Bit 11: Both types of drivers use this bit. The block driver uses it to tell MOS that it can support removable media. A diskette drive or a cartridge-based hard drive would be removable media. Bit 11 lets MOS use the media check function to determine if the media in the drive has changed. Character devices use this bit to let MOS know if they support the device OPEN/CLOSE calls. These calls should not be confused with the file/device function calls provided by MOS.

The OPEN/CLOSE calls can be used by the driver for special device initialization and/or resetting. For example, if the device is a printer, the OPEN call might have the driver send the printer a control sequence to set it to 12 characters per inch. The CLOSE call might set the printer back to its default state.

Bit 10 through 7: These bits are reserved.

Bit 6: Block drivers use this bit to indicate that the driver supports Getting/Setting a logical device. If a driver supports more than one logical device, MOS uses this call to tell the driver to set the new device or tell MOS which device is currently (get) set. On most 80286/80386-based machines, the 1.2M floppy disk drive can be drive A or drive B. MOS's floppy disk device driver controls both logical drives. See the generic IOCTL calls in the System Calls chapter for more information. Character device drivers do not use this bit and should set it to zero.

Bits 5 and 4: These bits are reserved.

Bit 3: This bit is set to one if the device is the current block device.

Bit 2: The driver should set this bit to one if it is the current NUL device. The NUL device is a bit bucket.

Bit 1: This bit tells MOS that the driver is the current standard output device. It is for character devices only. Block devices do not use this bit. BE SURE you see the section in this chapter on CONSOLE device drivers before attempting to create one.

Bit 0: Character devices set this bit to one to indicate if the device is currently the standard input device. Block drivers set this bit to one if they can support the generic IOCTL calls. See the System Calls chapter for more information on generic IOCTL.

THE PRIMARY ENTRY FIELD

All drivers provide a routine which must queue or store in some fashion the calls made to the driver. MOS calls the primary routine first. This routine saves the pointer in ES:BX for the secondary routine. When MOS gets control back from its call to the primary routine, it immediately calls the secondary routine.

THE SECONDARY ENTRY FIELD

MOS calls this routine to process its request. The secondary routine is responsible for retrieving the command code from the request header. This routine also sets the status information in the request header before returning to MOS. Note that all calls to the driver are FAR calls. Although MOS loads the driver in the SMP, the driver has no way of knowing before hand where the SMP is, so do not hard code any specific addresses.

THE NAME/UNITS FIELD

This field is used by character devices to store the name of the device. MOS uses the name as a way to access the device using the "handle" calls described in the System Calls chapter. Block devices may optionally set the first byte to the number of units the driver controls. MOS fills the value in when it gets the information back from the INIT call.

The Request Header

The request header is used by MOS to pass information to and get information from the device driver. This section will discuss the request header in depth.

The first 13 bytes are common to all headers. The header may have more information following it if the desired operation requires it. The header is passed to the primary entry routine in ES:BX. The first byte of the header is the total length of the header. This includes any data passed. Since this is a byte value, the maximum header length is 256 bytes. Byte two of the header is a logical unit indicator.

When a block device controls more than one logical device, this byte indicates from which logical device the service is requested. It is zero-based. For example, if the driver has two units, then this value could be 0 or 1. This byte has no meaning to character devices. The third byte is the function request code. Bytes 4 and 5 make up a word. The driver uses this field to return status information to MOS.

Next follow 8 bytes that are reserve by MOS. Should the request require any additional data, it is appended at the end of the request header.

The following table describes the function request codes.

Code	Name	Char	Block
0	Initialize	*	*
1	Media Check		*
2	Build BPB		*
3	IOCTL control string input	*	*
4	Read	*	*
5	Read w/o dequeue w/no wait	*	
6	Input Status	*	
7	Flush Input	*	
8	Write	*	*
9	Write and verify	*	*
10	Output Status	*	
11	Flush output	*	
12	IOCTL control string output	*	*
13	OPEN - if attrib bit 11 set	*	*
14	CLOSE - if attrib bit 11 set	*	*
15	Removable media		*
19	Generic IOCTL Request		*
23	Get Logical Device		*
24	Set Logical Device		*

The status field is zero upon entry to the device driver. The driver is responsible for passing the status of the device back to MOS upon completion of the requested function. This is how MOS gets errors back from the driver. The bits of this word are:

Bits 0 through 7: These bits define an error code if bit 15 is one.

Bit 8: This is the done bit. It is set to one when the driver returns.

Bit 9: This bit is set to one if the device is busy.

Bits 10 through 14: These bits are reserved by MOS.

Bit 15: The driver sets this bit to one if an error occurred while performing the desired function.

The following table lists valid error codes and their meanings:

Error	Description
0	Attempt to write on protected disk
1	Invalid logical unit
2	Device is not ready
3	Invalid command
4	Cyclic Redundancy Check error
5	Request structure length invalid
6	Seek Error
7	Media not known
8	Couldn't find sector
9	The printer is out of paper
10	Error writing to device
11	Error reading from device
12	General device failure
13	Reserved
14	Reserved
15	Invalid disk change

Device Driver Functions

The Init Function (0)

When MOS calls the driver to perform the init function, MOS uses 10 additional bytes in the data area of the request header. Besides the standard 13 bytes, MOS defines the extra 10 bytes as follows:

Byte 13: The number of units (block devices only). The driver sets this field.

Bytes 14-17: A double word pointer for the ending address of the resident portion of the driver.

Bytes 18-21: A double word pointer to a BPB pointer array. The driver sets this pointer field. If the device is a character device, MOS passes a pointer to the first character past the = sign from the DEVICE= statement in the CONFIG.SYS file. This string is terminated with a carriage return.

Byte 22: The first logical drive number for the device (block devices only). 0 = A, 1 = B etc.

The driver must do several things during the INIT call. Besides parsing the command line, if necessary, it must do any appropriate device initialization. If it's a block device, it must set the BPB pointer field in the additional data area for MOS and the number of logical units field (byte 14). It must set the ending memory address field. Finally, the driver must set the status word and put it into the request header.

Should the driver encounter errors during the INIT call, and for some reason not be able to load, it should set the ending address field to its CS:0. If it's a block driver, the unit number must be set to zero.

The Media Check Function (1)

This function call gives MOS the ability to determine if the media has changed. This helps in preserving data integrity on the media. Additional data areas appended to the request header are:

Byte 13: A media descriptor byte from MOS. The definition of the media descriptor is:

F0: 1.4M, 3-1/2"
F9: 720K, 3-1/2" or 1.2M, 5-1/4"
FC: 180K, 5-1/4"
FD: 360K, 5-1/4"
FE: 160K, 5-1/4"
FF: 320K, 5-1/4"

Byte 14: A return code. Possible values and their meanings are:

-1 - The media has changed.
0 - Can't tell if the media has changed.
1 - The media hasn't changed.

Bytes 15-18: A double word pointer to an ASCII string that is the previous volume label. This happens only if bit 11 of the attribute byte is one.

The driver must set the return byte and the status word in the request header.

The following table gives the conditions for the different return code values the driver must return to MOS. Conditions are tested in the order shown. Each test is made only if all conditions above it are false.

Condition	Code
A fixed disk w/non-removable media	1
If less than 2 seconds passed since last successful access	1
Drive has no change line	0
Drive change line not active	1
New BPB media byte not equal to passed byte	-1
Current volume ID NOT = Previous volume ID	-1
Current volume ID = Previous volume ID	0

Volume ID's are for the support of removable media. Should the driver support removable media and set the return byte to -1 indicating a media change, it must set the double word pointer in the request header to the address of the volume's ID. If the driver doesn't support a volume ID, it should set the pointer to the string "NO NAME",0.

The Build BPB Function (2)

The BIOS Parameter Block (BPB) is a structure that describes the media. When MOS calls the driver to build a BPB, it defines 9 bytes in the request header data area.

Byte 13: MOS's current media descriptor.

Bytes 14-17: Double word transfer buffer address.

Bytes 18-21: Double word pointer to BPB table.

The BPB describes the media to MOS. It provides a greater range of media types than the media descriptor byte.

BPB DEFINITION

Offset	Type	Description	Restrictions
0	word	bytes per sector	> = 32 power of 2
2	byte	sectors/cluster	> = 1 power of 2
3	word	# of reserved sectors	see below
5	byte	# of FAT's	2
6	word	# of root directories	Always 512
8	word	Total sectors and -1	see below
10	byte	media descriptor	see below
11	byte	sectors/fat	1 to 255
12	byte	total sectors/65536	see below
13	word	sectors/track	
15	word	number of heads	
17	word	hidden sectors and -1	< = 65535 bootable
18	byte	hidden sectors/65536	

The sector address of cluster #2 must be < = 65535, calculated as follows:

(reserved sectors + (# of FAT's * sectors/fat)) + ((# root entries * 32) / bytes/sector)

The last data cluster, which must be < 65280, is calculated as follows:

((total sectors - sector address of cluster #2) / sectors/cluster) + 1

Maximum values for 512 byte sectors:

128 sectors/clusters (64k clusters)
2 fats
FAT size 255 sectors
1 reserved sector
65280 clusters
512 root directory entries.

Media Descriptors and Format Types Supported:

FF	2 sides. 8 sectors/track
FE	1 side. 8 sectors/track
FD	2 sides. 9 sectors/track
FC	1 side. 9 sectors/track
F9	2 sides. 15 sectors/track
F0h	2 sides. 18 sectors/track
F8	Fixed disk

The Input and Output Functions (3,4,8,9,12)

This description applies to any of the listed I/O functions from the function table. For all the functions, MOS defines these additional fields in the request header data area.

Byte 13: MOS's media descriptor byte.

Bytes 14-17: Read/Write buffer pointer.

Bytes 18,19: Number of bytes or sectors transferred.

Bytes 20,21: Starting sector number (block devices only).

Bytes 22-25: Double word pointer to volume ID if an error 15 is returned.

After the operation is completed, the driver must set both the status word and the actual number of bytes or sectors transferred in the request header.

If the driver returns an error code of 15, it should set the pointer field in the request header to the correct volume ID.

The Read Without Dequeue Function (5)

When MOS calls this function in the device driver, the 13th byte of the request header is defined for the anticipated character. The driver must return the byte from the device and set the status word in the request header. Note that the character is NOT removed from the input queue. It must be removed by another READ function call. Character devices only.

The Status Functions (6,10)

MOS does not use any additional data in the request header for this call. This gives MOS the ability to determine if the device is busy or not. The driver performs the function, sets the busy bit and then the status word. The busy bit is interpreted differently for character devices depending upon the operation.

If the call is for a write status, the busy bit will indicate whether or not a write call can proceed or will wait for the bit to clear. For read status calls, the busy bit will indicate whether or not characters are queued for read calls. If no characters are queued, a read call will proceed through to the device. If there are characters in the input queue, the read call will get its character and return immediately.

The Flush Functions (7,11)

MOS defines no additional data in the request header for these calls. The driver sets the status word in the request header before returning. This call tells the driver to clear any I/O buffers. Any characters queued are lost.

The OPEN/CLOSE Functions (13,14)

Functions 13 and 14 require no additional data in the request header. These calls are optional. If bit 11 of the device attribute word is set, MOS calls these functions every time a file or device is opened and closed regardless of the device type. These functions can be used for whatever you desire. For example, a character device may send an initialization string to its device upon an OPEN request. It may send a different string for a CLOSE request. Block devices may use it for a buffering scheme.

The Removable Media Function (15)

If bit 11 of the device attribute word is set (1), this call can be used to determine if the device supports removable media. This function is called via a sub-function of function 44h. This function uses the busy bit of the status word in the request header to indicate if it supports removable media. If the bit is set (1), the media is not removable. If the bit is clear (0), the media is removable.

The Generic IOCTL Function (19)

Because Generic IOCTL supports several sub-functions, MOS defines several additional bytes in the request header data area.

Byte 13: Major function code

Byte 14: Minor function code

Bytes 15,16: Contents of the SI register

Bytes 17,18: Contents of the DI register

Bytes 19-22: Pointer to an info packet

Generic IOCTL deals with logical devices. It provides for reading and writing tracks on a logical device, formatting a logical device and getting/setting logical device attributes. Check the Generic IOCTL function call in the System Calls chapter for more information. The sub-functions defined in that section must be supported.

All IOCTL type calls made to MOS are passed to the driver after the request header has been built. The driver must maintain its own track layout table. The track layout table describes how the tracks look on the media . This includes the number of sectors per track and the size, in bytes, of each sector. The FORMAT command makes extensive use of Generic IOCTL.

The Get/Set Logical Device Functions (23,24)

When a block device driver supports logical devices, it usually means that the driver controls multiple units. For instance, with a 1.2M floppy drive, the MOS floppy driver can access the drive as either A or B. When an application changes from one logical device to the other, MOS calls these functions. For the GET function, MOS places a disk ID (0=A, 1=B, etc.) in the unit field of the request header. The driver puts the disk ID of the last device accessed in the unit field of the request header. For the SET function, MOS sets the desired unit code in the unit field of the request header. The driver replaces that with the code of the last unit accessed.

Device Driver Design Considerations

When developing a device driver, being able to add and remove it "on the fly" (i.e. without rebooting) using ADDDEV and REMDEV can speed up the development process significantly. Be aware, however, that when using ADDDEV, any vectors which the driver sets will only take effect for the task doing the loading. This can sometimes cause problems since this makes the driver only partially task-specific. When a driver is loaded from a DEVICE= statement within CONFIG.SYS, all hooked vectors will be global.

A driver loaded into the SMP with ADDDEV will be linked into the global list of device drivers. Therefore, while it will only be accessible to the current task through any interrupt vectors, it could be opened by its device name by any task.

ADDDEV can also be used to load a driver in a task-specific manner. The driver is loaded into task memory much the same way that a TSR loads. In addition, the driver's header is linked into a list which is private to that task. A task specific is searched before the global device driver list.

If a device driver should not be allowed to be loaded in a task-specific manner, the driver can call Extended Services function 25H during its initialization process. If the AX return value is 0 then the driver should report the condition and abort the installation.

Note that if you are designing a driver which will make Extended Services calls then the initialization procedure should obtain a vector to these services as described in Chapter 13. This is more reliable and more efficient than issuing a software interrupt call to INT 38H or INT D4H.

Effects on Task Switching

During the initialization of a device driver which is loaded through CONFIG.SYS, there is no concern about a task switch being caused from a timer-tick interrupt or from a system call being made. When CONFIG.SYS is being processed, there are no other tasks.

Likewise, when a driver is loaded into the SMP with ADDDEV, the initialization code can be designed to presume that no task switches will occur from a timer-tick interrupt. Timer-tick interrupts which occur when the current point of execution is outside of the task space will not invoke a task switch.

Once a driver has been loaded into the SMP and initialized, its run-time code, the code which executes in response to function calls being made to the driver, will also never be time-sliced. However, this run-time code could make calls to the MOS kernel for services and end up having a task switch occur. When calls are made for services to INT 10H, 14H, 16H, 17H, 21H, and to MOS's Extended Services handler, a task switch may occur.

There can be situations where a driver will also want to explicitly call Extended Services function 7 to give up the processor and allow a task switch. One example of this would be the $SERIAL.SYS driver. When an attempt to output more data is made while the driver's buffer is full, an explicit suspend call is made. When some room has cleared within the buffer, the new output request will be processed and the function call will return to the caller. Writing to the $PIPE.SYS driver when its full or reading from it when its empty is another example of this.

A device driver which allows itself to be sliced must be coded with re-entrance in mind. Another call may be issued to this same driver by another task while an existing call is in progress. To manage this, all data which needs to be task-specific should be kept on the stack or in a task table.

When re-entrance is allowed it becomes important in such cases to clearly analyze which data must be task-specific and which should be global. The double word pointer to the request header is one example of this. Any driver code which must access the request header (e.g. to set the return status) must use the task-specific pointer.

As stated above, if an INT 8H occurs while a device driver call is being processed then no task switch will be done. This test is done by checking the point of execution when the interrupt occurred. A timer interrupt task switch is only done when the current point of execution is within the task space. One consequence of this is that a device driver loaded using ADDDEV's task-specific option can be sliced.

Using a Task Table

When a device driver must keep track of a large amount of task specific data, using a task table becomes a much more attractive choice than keeping data on the stack. There could also be cases where the data for all tasks must be accessible at all times. This would not be possible when stack storage was used.

You may want to construct a table design for a fixed number of tasks or you may want to let the user specify the maximum size on the parameter line following the "DEVICE=drvrname.sys" within CONFIG.SYS. Regardless, it is important to provide a graceful response when a new task requests the driver's services and its table is full.

Regarding the key field to use to identify each task in such a table, the most straight-forward method is to store the task's TCB segment address. If you decide to use the task's ID number instead, be aware that if you decide to index the table by its ID, the situation could exist where there are only two tasks in a system: task 0 and task 99. The task's ID number is stored in the TCBID field. Access to the TCB data structure is covered in Chapter 13.

Since tasks are a dynamic entity within PC-MOS, the maintenance of a task table requires the ability to detect when a task has been removed so its storage space can be marked as available. This can be handled through the Task Unregistration process, as described in Chapter 6.

Terminal Interface

Purpose

This interface provides terminal support for MOS. These are character devices.

Functions

The terminal device driver allows MOS to efficiently make a terminal "look" like a standard PC monitor and keyboard.

In MOS, there are two basic types of terminals. The first type is the PC-type terminal, like a Kimtron KT-7 or Wyse 60™. The second type is a standard ASCII terminal, like the ADDS Viewpoint or the Televideo® 925. The PC-type terminals send scan codes just like the keyboard on the PC. The ASCII terminals send the corresponding ASCII code for each key rather than a scan code.

We have designed an ASCII terminal driver which is assembled with an include file. The include file provides terminal specific information using a set of standard labels. The main body of the driver provides the basic functionality of the driver. So, all the developer should need to do is create the include file, and assemble the two, to produce the needed driver. The .ASM files for both the PC-type terminal and standard ASCII terminal and sample include files are on the accompanying diskette.

The terminal device driver provides two classes of functions. The first class is the defined functions the driver must support because it is a standard character device driver. The second class are those necessary to support the actual terminal.

A far pointer to the entry point of the function handler for this second class of functions is obtained by opening the device, placing the handle in raw mode, and reading four bytes. The four bytes read should be placed into a doubleword pointer field and used to make far calls to the driver's secondary function handler. Chapter 13 illustrates this technique in more detail.

The first class of functions are defined below:

Function	Purpose
0	Driver Initialize
1	Media Check
2	Build BPB
3	IOCTL Input
4	Read Entry Point

NOTE: Build BPB is not used for character devices.

The second class of functions needed for actual terminal support will have a more detailed discussion later. Here is a summary of the functions:

Function	Purpose
0	Register a port
1	Read keyboard scan code
2	Clear current scan code
3	Set terminal mode
4	Set cursor type
5	Set physical cursor position
6	Scroll up
7	Scroll down
8	Write character & attribute
9	Set palette

10	Write TTY
11	Write string
12	Re-Display window
13	Write char @ row,col
14	Read char @ row,col / page
15	Get screen (master console only)
16	Send char to terminal prn
17	Send string to term prn

Initialization

MOS loads the driver into the System Memory Pool (SMP) and calls the driver to initialize it. After the driver returns from the init call MOS does nothing with the driver until the ADDTASK command is executed to add a terminal related task. ADDTASK opens the device via function 3D. It reads four bytes, via function 3F, to get the driver's entry point. ADDTASK stores the entry point in the task's TCB. ADDTASK then closes the device driver.

The first request the MOS kernel makes to the driver for terminal initialization is function 0. Once the terminal has received the "Key Ready" flag from the serial driver, the driver's terminal support functions are called as needed. It is the driver's responsibility to translate the requests to the appropriate command codes to control the terminal.

Special Considerations

ASCII terminal drivers are responsible for translating received ASCII characters into corresponding make/break codes to faithfully emulate the PC keyboard. See the sample include files for terminal specific requirements.

The basic differences between PC-type terminals and ASCII terminals are:

1. The PC-type supports 25 lines. Most ASCII terminals do not.

2. The PC-type keyboard sends make/break codes and no translation is necessary.

3. The character display attributes of the PC-type are the same as a PC Monochrome display.

Several functions have DS:SI pointing to a substructure in the Task Control Block (TCBDDT). DDT stands for Device Driver Terminal.

The structure's definition is:

```
TCBCONDD    DD      0           ; Reserved
TCBDLOFS    DW      0           ; Offset - pointer to screen image
TCBDLSEG    DW      0           ; Segment - pointer to screen image
TCBDCOFS    DW      0           ; Cursor's offset within page
TCBDCOL     DB      0           ; Number of displayed cols on screen
TCBPORT     DW      0           ; Port number. 0 = none
TCBBAUD     DD      0           ; Actual transmission speed
            DB      19 DUP(0)   ; Used by DDT's for scratch area
```

The next few pages give a detailed description of the terminal related functions the driver must support.

Function 0 - Register a Port

Registers upon entry:

AH = 00h
DS:SI = points to TCBDDT for this terminal

Return Values:

AH = 00h if no errors and
AL = Bit Flags. If bit 6 is set the device supports graphics, and graphics calls are allowed. If bit 7 is set, terminal is the master console. Bit 4 of AL indicates the driver supports Hercules™ monochrome graphics if the bit is set (1).
ES:BX = points to "key ready" flag. Flag is non-zero if characters (scan codes) are buffered.

Errors Returned:

AH = 1 - bad port number
AH = 2 - bad baud rate

Description:

For serial terminals, this function calls the serial device driver with AH = 13 to "register" the port. It returns to its caller the address of the "key ready" flag is ES:BX.

Function 1 - Input Keyboard Scan Code

Registers upon entry:

AH = 01h
DS:SI = pointer to TCBDDT for this terminal

Return Values:

CY = 1 (carry flag set) if the terminal is using the new carrier detect feature for automatic reboot. *
AL = Bit 7: 1 = key break, 0 = key make or repeat. Bits 0 - 6 key scan code 01h - 07Dh.

* this feature was introduced in release 4.00 of MOS

Errors returned:

None

Description:

This function reads a scan code non-destructively. If the driver is for a PC-type terminal, a make/break code is read from the serial driver. If the driver is for an ASCII type terminal, the driver must return the appropriate make/break sequence and therefore may not call the serial driver for a character each time this call is made. Check the sample ASCII driver file.

Function 2 - Clear Current Scan Code

Register upon entry:

AH = 02h
AL = bit 7: 1 = clear buffered scan code, ignored if 0. Bits 0 - 6 ignored.
DS:SI = points to TCBDDT

Returned Values:

None

Errors Returned:

None

Description:

This call tells the driver to clear the current scan code. This allows the driver to get ready for reading the next scan code. Implement the clearing any way that's necessary.

Function 3 - Set the Terminal Mode

Registers upon entry:

AH = 03h
AL = 00h, 01h - 40 x 25 text mode
AL = 02h, 03h, 07h - 80 x 25 text mode
AL = 04h, 05h - 320 x 200 graphics mode (if applicable)
AL = 06h - 640 x 200 graphics (if applicable)
DS:SI = points to TCBDDT

Returned Values:

None

Errors Returned:

None

Description:

If the terminal supports these modes, this function allows the driver to set the terminal in the appropriate mode. When the mode is changed, the screen should be cleared and the cursor set to 0,0, the top left-hand corner of the screen.

Function 4 - Set Cursor Type

Registers upon entry:

AH = 04h
CH = beginning scan line
CL = ending scan line
DH = row
DL = column
DS:SI = pointer to TCBDDT

Return Values:

None

Errors Returned:

None

Description:

Function 4 tells the terminal to display different sized cursors, such as block or underline.

Function 5 - Position the Cursor

Registers upon entry:

AH = 05h
DH = row
DL = column
DS:SI = points to TCBDDT

Return Values:

None

Errors Returned:

None

Description:

This function causes the driver to send the command sequence to position the cursor at a particular row and column on the terminal screen.

Function 6 - Screen Scroll Up

Registers upon entry:

AH = 06h
AL = number of lines to scroll. 0 to clear region
BL = attribute to use on blank lines
CH = starting row
CL = starting column
DH = ending row
DL = ending column
DS:SI = points to TCBDDT

Return Values:

None

Errors Returned:

None

Description:

Function 6 forces the terminal to scroll the window defined by the CX and DX registers up the number of lines in AL. If AL = 0, the entire window is erased.

Function 7 - Scroll Screen Down

Registers upon entry:

AH = 07h
AL = number of lines to scroll. 0 to clear region
BL = attribute to use on blank lines
CH = starting row
CL = starting column
DH = ending row
DL = ending column
DS:SI = points to TCBDDT

Returned Values:

None

Errors Returned:

None

Description:

Function 7 forces the terminal to scroll the window defined by the CX and DX registers down the number of lines in AL. If AL = 0, the entire window is erased.

Function 8 - Write Character and Attribute

Registers upon entry:

AH = 08h
AL = character to write
BL = attribute to write
CX = number of times to write
DH = row
DL = column
DS:SI = points to TCBDDT

Returned Values:

None

Errors Returned:

None

Description:

Function 8 writes the specified character and attribute to the current screen page at the specified row/column. The character/attribute is written CX times.

Function 9 - Set Color Palette

Registers upon entry:

AH = 09h
BL = byte as stored in crt_palette for BIOS
(BL and 1Fh = background color)
(BL and 20h = palette 1 if non-zero, otherwise palette 0)
DS:SI = points to TCBDDT

Returned Values:

None

Errors Returned:

None

Description:

This emulates the BIOS set palette function.

Function 10 - Write TTY

Registers upon entry:

AH = 0Ah
AL = Character to write
DH = row
DL = column
DS:SI = points to TCBDDT

Returned Values:

None

Errors Returned:

None

Description:

Function 10 emulates the ROM-BIOS write TTY function. The carriage return, line feed, bell and backspace characters are treated specially.

The carriage return moves the cursor to the beginning of the current line. A line feed moves the cursor down one line. If the cursor is on the last line, the entire screen is scrolled up one line. The bell character sounds a beep. Backspace moves the cursor back one space non-destructively. Attributes are not affected.

Function 11 - Write a String of Characters

Registers upon entry:

AH = 0Bh
CX = number of words to write
DH = row
DL = column
ES:DI = Address of string to write
DS:SI = points to TCBDDT

Returned Values:

None

Errors Returned:

None

Description:

Function 11 gives the driver the ability to write a string of characters in one call.

This function is used for screen updates in both graphics and text modes. In text mode, characters are expected to be stored in the same order as video memory - character/attribute, etc.

In graphics mode, the string consists of words of data to sequentially write to video memory. Attributes are irrelevant. DH is the scan line address and DL is the byte address within the scan line. This function always uses words.

Function 12 - Re-Display Window

Registers upon entry:

AH = 0Ch
AL = 0 - re-display top lines of screen
AL = 1 - re-display bottom lines of screen
CX = Number of words to transfer
DS:SI = points to TCBDDT

Returned Values:

None

Errors Returned:

None

Description:

For terminals with 24 or less rows, this function allows the driver to re-display rows 0 through N (the top lines) or rows 24-N through N (the bottom lines), where N = Last row number - 1. N must be > = 12. CX is the number of words in the screen to move.

Function 13 - Write Character

Registers upon entry:

AH = 0Dh
AL = character to write
CX = write count
DH = row
DL = column
DS:SI = points to TCBDDT

Returned Values:

None

Errors Returned:

None

Description:

Function 13 writes only the character in AL to the specified row/column on the current page. Attributes are not affected.

Function 14 - Read Character/Attribute

Registers upon entry:

AH = 0Eh
DH = row
DL = column
DS:SI = points to TCBDDT

Returned Values:

Not applicable

Errors Returned:

Not applicable

Description:

This function applies to the main console only.

Function 15 - Get Screen

Registers upon entry:

AH = 0Fh
CX = word count to get
DS:SI = points to TCBDDT

Returned Values:

Not applicable

Errors Returned:

Not applicable

Description:

Function 15 applies to the main console only.

Function 16 - Send Character to Terminal's Printer

Registers upon entry:

AH = 10h
AL = character to print
DS:SI = points to TCBDDT

Returned Values:

AH as in INT 17h call.

Errors Returned:

Must emulate INT 17h status return values. The return bits are:

0: 1 = time out
1: unused
2: unused
3: 1 = I/O error
4: 1 = printer selected
5: 1 = out of paper
6: 1 = acknowledge
7: 1 = idle

Description:

If the terminal supports remote printing via an attached printer, this function is used to access the printer. The driver must emulate the return codes that the INT 17h logic returns. If printer support is not included within the driver, it should immediately return with AH = 1.

Note that some serial workstations do not provide printer status feedback.

Function 17 - Send a String to Terminal's Printer

Registers upon entry:

AH = 11h
CX = length of string in bytes
ES:DI = points to string to print
DS:SI = points to TCBDDT

Return Values:

Same as function 16.

Errors Returned:

AH = same as INT 17 logic if the driver couldn't print the whole string. DI points to the first character not printed and CX is the number of characters NOT printed.

Description:

Function 17 gives a string print capability to the device driver if the terminal supports transparent printing. If printer support is not included within the driver, it should immediately return with AH = 1.

Note that some serial workstations do not provide printer status feedback.

Serial Device Interface

Purpose

This provides MOS with a standard communications interface for serial devices. It is most often used to support terminals and serial printers, although it is not limited to these areas.

Functions

The following table is a summary of the driver's functions. A more in-depth discussion of the driver's functions will follow. It is NOT recommended that the developer extend the numbers of functions listed, in order to maintain system compatibility. However, we do want to hear your suggestions and needs, in order to improve our products.

The number in parentheses beside the function number is the MOS extended function number. It has the high-order bit of the AH register set. The driver should be able to handle both. There are two numbers for each function to allow for ROM-BIOS Interrupt 14h expansion and driver expansion. Currently, both numbers will cause transfer of control to the same function.

Function	Purpose
0 (80h)	Port initialization
1 (81h)	Write a character
2 (82h)	Read a character
3 (83h)	Return port status
4 (84h)	Extended port init
5 (85h)	Change port protocol
6 (86h)	Driver ID
7 (87h)	Send RS232C "Break"
8 (88h)	Non-destructive read
9 (89h)	Reset all buffer pointers
10 (8Ah)	Report number of input chars
11 (8Bh)	Disable the port
12 (8Ch)	Return current port settings
13 (8Dh)	Register terminal use of port
14 (8Eh)	String output
15 (8Fh)	String input
16 (90h)	Link to another serial driver
17 (91h)	Write Modem Control Register
18 (92h)	Return Driver Description
19 (93h)	Selective Buffer Flush
20 (94h)	Output Queue Check
21 (95h)	Write a character (with time out)
22 (96h)	Read a character (with time out)
23 (97h)	Declare port ownership

Initialization

The driver is responsible for providing default values for input and output buffer sizes, transmission speed, number of stop bits, parity, line protocol and the number of ports it is to control. These are overridden by any options passed from the command line. The options are the responsibility of the developer. Port numbers read from the command line are one based meaning that the first port defined becomes logical port # 1 for that driver. If this is the first serial driver installed, this will correspond to COM1.

However, if multiple serial drivers are chained, this will only be true for the first driver. Successive drivers will have their logical port numbering adjusted by the number of port numbers which precede.

Before the driver returns to MOS from the driver init call, it must issue a function 6 (86h) call via interrupt 14 to see if another serial driver is loaded before it. If function 6 indicates a driver is loaded, the init code should issue the link function call. If the function 6 indicates no other driver is loaded, the driver should set the INT 14 entry point at 0:50 and return from the initialization call.

Once the driver is loaded and its initialization routine called by MOS, all ports are preset with default values. All the ports are "disabled" until an application or other device driver calls the serial driver to initialize the specific port.

Because of this, any calls other than a register (13), or init (0,4) will return an "Invalid port" error to the caller. This is a 0FFh in the AH register. Once the port is "initialized" or "registered", the application should interpret the return of a 0FFh in the AH register as a function specific error. Consult the definition of the specific function for the error interpretation.

When a terminal task is added, the terminal driver calls the serial driver with the following functions, in order: 13, 4, 5. This sequence tells the serial driver to set up the port for use. For character I/O the terminal driver may use functions 1 and 2 or functions 14 and 15. The string functions are preferred for speed.

When MOS gets control via a hardware interrupt or an INT call, it checks the "Key Ready" flag returned by function 13 to see if there are any characters in the input buffer. If there are, the terminal driver is called to remove them. To remove characters from the input buffer, the terminal driver currently issues a function 8 call, then removes the character from the buffer via function 2. All this occurs for each terminal related task. See the section on Terminal Device Drivers if you need further information.

Special Considerations

Should the driver do any polling logic awaiting incoming characters or waiting to send a character or for any reason, the polling logic should allow for giving up the CPU to enhance system throughput. For recommended ways to yield CPU control, see function 7 of MOS's Extended Services in the System Calls chapter and the programming example in Chapter 13. If possible, the driver should be re-entrant.

If your driver is for an intelligent serial card, it is possible to operate with very low overhead to the system CPU. Either the operating system on the smart card or the driver must set the "key ready" flag. It should be the card operating system.

Most smart serial cards use dual-port RAM for communication between the CPU's. The driver and the card should allow flexibility in setting the segment address of the dual-port RAM. MOS can use the extra segment addresses above the video RAM segment address for itself. On a system using EGA, it can be difficult to find an extra segment for the smart card, so allow for flexibility.

Not all the functions are used in the MOS environment, but the driver should implement all of them for compatibility.

Finally, the smart card operating system should be easily upgraded. Some cards download their operating system from disk via the device driver. Others have theirs on an EPROM. As MOS matures, it may allow for the off-loading of some of its terminal support functions to the smart card's operating system.

Provisions have been made for multiple serial drivers sharing INT 14H. See Function 16 for details.

The following pages provide an in-depth discussion of the driver's functions and register usage. The port number specified in the DX register for the functions is zero-based.

Function 0 (80h) - Initialize the Port

INT 14 Entry Registers:

AH = 0 (80h)
AL = bps, data length, parity
DX = port number.

Bit definitions for the AL register are:

7 6 5	4 3	2	1 0
baud rate	parity	stop bits	data len.
000 - 110 bps	00 - none	0 - 1	10 - 7 bpc
001 - 150 bps	01 - odd	1 - 2	11 - 8 bpc
010 - 300 bps	11 - even		
011 - 600 bps			
100 - 1200 bps			
101 - 2400 bps			
110 - 4800 bps			
111 - 9600 bps			

Return Values:

AH = Port Status (defined under Function 3).
AL = Modem Status

Errors Returned:

None.

Description:

This function initializes the port for data transmission. It enables a port disabled via function 11. It allows for dynamically changing port parameters.