Microsoft Multitasking MS-DOS Product Specification

DEVICE DRIVERS

October 30, 1984

ABSTRACT

This document describes the services available to MS-DOS 4.0 device drivers and the requirements they must meet. It assumes that the reader is familiar with the operation of device drivers under MS-DOS 2.0.

1. INTRODUCTION

This document discusses the MS-DOS 4.0 device driver architecture, followed by a summary of the new MS-DOS service routines and the changes in the device header and request packet formats. Details of these are then covered with particular attention to the changes in the console driver (CON) and block device drivers.

This document is one of a series of related documents. They are:

- *Microsoft Multitasking MS-DOS Product Specification OVERVIEW*
- *Microsoft Multitasking MS-DOS Product Specification DEVICE DRIVERS*
- *Microsoft Multitasking MS-DOS Product Specification SYSTEM CALLS*
- *286 and 8086 Compatibility*
- *Microsoft Multitasking MS-DOS Product Specification INTRODUCTION*
- *Microsoft Multitasking MS-DOS Product Specification MEMORY MANAGEMENT*
- *Microsoft Multitasking MS-DOS Product Specification DYNAMIC LINKING*
- *Microsoft Multitasking MS-DOS Product Specification SESSION MANAGER*

2. OVERVIEW

Existing MS-DOS 2.0 device drivers are typically synchronous and non-interrupt-driven. Since MS-DOS 2.0 is a single-task operating system this presents no problem. The program cannot proceed until the I/O is done, so it is acceptable for the device driver to hold the CPU until the I/O is complete.

MS-DOS 4.0, on the other hand, is a multitasking operating system. It is important that device drivers be interrupt-driven and written in such a fashion that they surrender the CPU while they are awaiting device completion. The DOS can then assign the CPU to other tasks that are not waiting on I/O.

Although interrupt-driven device drivers are highly desirable for performance reasons, polling device drivers are still usable. MS-DOS 2.0 device drivers are compatible with MS-DOS 4.0 under the following conditions:

1) If they have hardware interrupt handlers, the handlers do not enable interrupts; that is, no nested interrupts can be handled in an MS-DOS 2.0 device driver.

2) They are not the CON device driver. Existing MS-DOS 2.0 CON drivers cannot be used because they do not support the "screen image" facility, described in Section 9.2.

Non-interrupt device drivers will operate as they did under MS-DOS 2.0 but will slow system performance. For best results, all device drivers should be rewritten to use the features provided in MS-DOS 4.0.

3. NEW FEATURES

There are three major areas of change for MS-DOS 4.0 device drivers:

1) A new device driver architecture is defined. The new architecture supports multiple (parallel) I/O requests and interrupt-driven non-polled operation. Since many different tasks may be running simultaneously, several of them might make an I/O request at the same time. The MS-DOS 4.0 device driver architecture supports a queuing mechanism that allows the device driver to work on more than one request at one time, and to choose the request it will service next from the list of outstanding requests.

The interrupt-driven architecture allows the device driver to surrender the CPU while it is waiting on an I/O operation, and regain it later when the operation completes.

2) The MS-DOS 2.0 CON driver has been split into two parts: Keyboard Input and Console Output. Both the input and output drivers support the interrupt-driven architecture described above. Both have some additional changes for MS-DOS 4.0.

   The Console Output driver is changed in two areas. It must take on a few duties previously handled by the DOS, such as tab expansion. It must also support a "multi-screen" capability. This means the ability to save the contents of the screen into a DOS-supplied RAM buffer, followed by the restoration of some previously saved screen image from another RAM buffer.

3) Much of the work described here is logically a function of the device driver but is not device dependent. MS-DOS 4.0 provides a package of routines to do the majority of this new work. If these routines are properly employed, MS-DOS 4.0 device drivers are no more complex to create than MS-DOS 2.0 drivers.

4. NEW DRIVER ARCHITECTURE

As described in the previous section, the MS-DOS 4.0 device driver architecture is designed to support the queuing of multiple I/O requests to be serviced in the most efficient order. A disk device driver, for example, can queue several read operations and use some type of algorithm to minimize the traveling of the heads across the disk. (The DOS makes available an implementation of just such an algorithm.) MS-DOS 4.0 drivers should use two DOS-supplied routines called **ProcBlock** and **ProcRun** so that the CPU can be reassigned while I/O is in progress.

MS-DOS 4.0 device drivers are called from the DOS with an I/O packet that describes the requested I/O operation. They may return to the DOS before the operation is complete; the DOS will take care of suspending any tasks that need the results of the operation.

Old-style device drivers will likely poll the device waiting for it to complete its task; this is acceptable, but seriously restricts I/O throughput. This also wastes the CPU time spent in the driver wait loop—it cannot be given to other tasks that are ready to run. Simulating interrupt-driven operations for non-interrupt-driven devices will be described later in this document.

4.1. I/O Handling

There are two basic methods for handling interrupt-driven block device I/O. In the preferred method, the DOS calls the device driver strategy entry point with the address of the I/O request packet. The **Strategy** routine checks the validity of the I/O request. If the request is valid, the **Strategy** routine places the request on a work queue for this device, using the DOS **PushRequest** or **SortRequest** functions. These DOS functions allow the I/O request to be placed on the work queue in either a first-in, first-out (FIFO) or sorted order.

If the device is currently idle, the **Strategy** routine starts up the device, usually by calling a **Start** routine in the device driver. The **Strategy** routine then returns to the DOS with the I/O request marked incomplete (i.e., the STATUS done bit is reset). The DOS puts the process to sleep until the I/O request has been completed.

When the device interrupt occurs, the interrupt handler must set the return status in the request packet and call the DOS **DoneRequest** routine that marks the request as complete (STATUS done bit set) and wakes up the waiting process. The device handler gets the next I/O request using the DOS **PullRequest** function. If there is an I/O request, the interrupt handler starts the request and exits the interrupt handler.

An alternate method allows the process to remain under control of the device driver until the I/O request has been completed. This is done by calling the DOS **ProcBlock** function to put the process to sleep, instead of returning an incomplete status from the **Strategy** routine. The

interrupt handler then calls the DOS **ProcRun** function to wake up the process instead of calling the DOS **DoneRequest** function. The process may then set the return status in the request packet, set the STATUS done bit, and start the next request prior to returning from the **Strategy** routine.

The easiest way to understand this device driver architecture is to envision two independent and parallel processes: the **Strategy** routine and the **Interrupt** routine. The **Strategy** routine validates requests, puts them on the work queue, and waits for them to eventually complete. The **Interrupt** routine examines the results of the operation, handles error retries, and indicates that the request is done.

Note that **Strategy** initiates the I/O only if the device had been inactive; if the device is working on some previous request, the **Interrupt** routine will issue this new request when it reaches the head of the work queue. The **Strategy** code which checks for active I/O to consider calling **Start** must lockout interrupts to avoid races with **Interrupt**.

4.2. Handling Multiple I/O Requests

To this point, this discussion of the device driver architecture has been from the viewpoint of a single I/O request. There will actually be multiple activations (or instances) of the **Strategy** routine, one for each request on the work queue. If process *X* calls the DOS to do some I/O, the DOS will in turn (running as process *X*) call the driver's **Strategy** routine. When **Strategy** calls **ProcBlock**, it is the process *X* that is being put to sleep. When process *X* calls **ProcBlock**, the CPU might be assigned to process *Y* which could, in turn, issue an I/O request and end up in that same **Strategy** routine, also calling **ProcBlock**. In this case, there are two instances of the **Strategy** routine active.

When an I/O operation is complete, **Interrupt** automatically wakes up the appropriate instance by supplying the request block address as an argument to **ProcRun**. Service need not be on a FIFO basis, because **Strategy** may choose to sort requests.

The **Strategy** routine may assume that it will not be preempted by other task-time instances, but it must protect itself against its own **Interrupt** routine. It should clear the interrupt flag when checking if the device is active and when examining the device queue. The **Interrupt** routine will only be preempted by other higher priority interrupts, and only if it re-enables interrupts.

The mechanism for queuing and processing requests is illustrated in the figure on the following page.

MS-DOS 4.0 Driver Model

Strategy (Called by the DOS)	**Start** (Called by **Strategy** and **Interrupt service routines**)	**Interrupt Service** (Entered by hardware interrupt)
Called by the DOS with request. Verify request and put on queue. If I/O is not already active call **Start**. Return to MS-DOS.	Issue first request on queue to device. Return to caller.	
	Issue first request on queue to device. Return to caller.	**Device Interrupts:** Analyze device status. If error then call **Start** to retry or set error flag. Remove request from queue. Issue **DoneRequest** on request header. Call **Start** to issue next request, if any. IRET

Note that the vertical axis represents time and that time increases from top to bottom of the figure.

5. MS-DOS SUPPLIED SERVICES

As discussed previously, many of the functions of an MS-DOS 4.0 device driver are related to its interface with the DOS rather than the device. It is sensible for the DOS to make available much of this interface code for the use of the device drivers.

As in MS-DOS 2.0, each driver has an "initialize entry point" that is called during system boot-up. When MS-DOS 4.0 calls the initialize entry point, it supplies a pointer to a subroutine that provides these services. A parameter to the subroutine indicates which of the subfunctions described below is desired.

This section briefly describes the DOS routines and their purpose. This description is intended to facilitate the reading of the remainder of this document.

- **ProcBlock** (*ProcBlock-code, time-limit*)

  This routine is called to suspend execution of this instance of the device driver. The DOS will not immediately return from **ProcBlock**; it removes the current task from the run queue and starts executing some other task. The *ProcBlock-code* is an arbitrary 32-bit value, but conventionally a process calls **ProcBlock** on a particular address, such as the address of a command block or buffer header. The process is reactivated and **ProcBlock** returns when **ProcRun** is called with the same *ProcBlock-code* or when the *time-limit* expires. **ProcBlock** sets the condition codes to indicate a normal call of **ProcRun** or an expiration of **ProcRun**. A *time-limit* value of 0 means to call **ProcBlock** indefinitely until awakened. **Procblock** can only be called by the task-time portion of a device driver.

- **ProcRun** (*ProcBlock-code*)

  This is the companion routine to **ProcBlock**. When called, it awakens ALL processes that were blocked for this particular *ProcBlock-code*. This makes it easy for the **Interrupt** routine to know what code to issue to waken those waiting for a particular request—it just wakes every process calling **ProcBlock** on that request block.

  ProcRun returns immediately to its caller; the awakened process(es) will be run at the next available opportunity. **ProcRun** is often called at interrupt-time.

- **DoneRequest** (*request*)

  This routine can be used to mark a request block as DONE and to perform **ProcRun** for any processes that may be calling **ProcBlock**. The driver should set any error flags and codes in the status word before calling the routine.

- **PushRequest** (*queue-head, request*)

  PushRequest adds the current device request packet to the list of packets to be executed by the device interrupt routine. The device driver should add all incoming read/write requests to its request list. The driver should then determine whether the interrupt-time execution thread is active; and if not, it should call the **Start** routine. Since the device may be active at this point, the caller must have turned off interrupts before calling this routine (otherwise a window exists in which the device finishes before the packet is put on the list).

- **PullRequest** (*queue-head*)

  PullRequest pulls the next waiting packet address from the selected device queue. If there is no packet (such as when characters are typed on the keyboard before they are read), then the zero flag is set on return.

  Typically a device driver uses **PushRequest/PullRequest** to maintain a request queue for each of its devices/units. An empty queue is just a 0:0 DWORD. Queue elements are chained onto this pointer. The device driver must allocate and initialize the queue header DWORD to 0 before using **PushRequest**.

- **SortRequest** (*queue-head, request*)

  This routine can be used by block (disk) device drivers to add a new request to their work queue. This routine adds the request to the queue and sorts the request by starting sector number. Sorting reduces the length of head seeks and speeds disk throughput.

- **ConsInputFilter** (*input char*)

  This routine is called by the keyboard interrupt handler to allow the DOS to scan the input stream for special characters. It returns an indication of whether the character should be queued or discarded. See Section 9.1, "Keyboard Interrupts," for more details.

- **Signal_SM** (*screen-number*)

  Arranges to have a screen switch signal sent to the screen manager. *Screen-number* is the number of the screen to be switched to. See Section 9.1, "Keyboard Interrupts," for more details.

- **GetDOSVar** (*VarNumber, Index, Size*)

  Returns the address or value of an internal DOS variable. *VarNumber* selects the variable to be returned, *Index* selects an item in a list or array, and *Size* indicates the expected size of the item.

- **QueueFlush** (*char-queue*)

  The character queue structure pointed to by the argument is initialized.

- **QueueWrite** (*char-queue, char*)

  The character passed is added to the character queue structure. An error indication is given if there is no room in the queue.

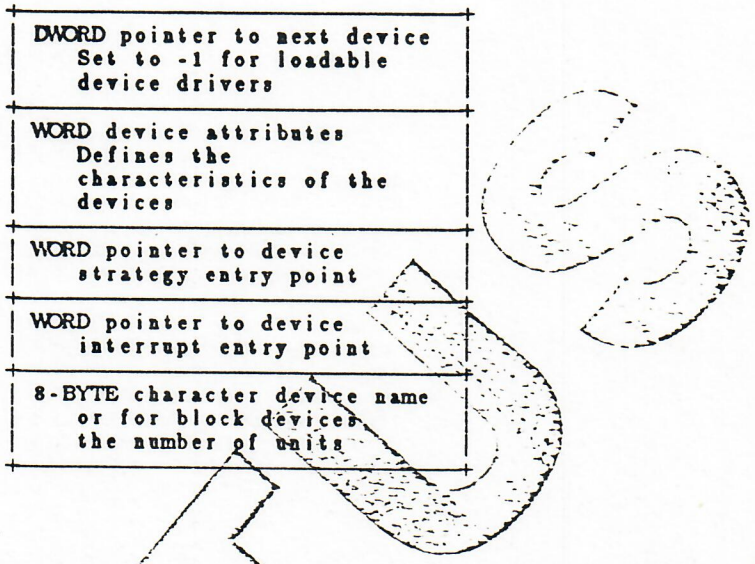
- **QueueRead** (*char-queue*)

  The character at the beginning of the character queue structure is returned and deleted from the queue. An error indication is given if there are no characters available.

6. DEVICE HEADER

The format of an MS-DOS 4.0 device header remains essentially the same as in MS-DOS 2.0. The device header retains the two entry points defined in MS-DOS 2.0, but they are treated as being identical and are called in sequence. This is maintained for backward compatibility. If bit 5 (the MS-DOS 4.0 bit) of the attribute word is set, the second entry point should be set to zero and will not be called.

MS-DOS 4.0 DEVICE HEADER



6.1. Attribute Word

The attribute word changes slightly in MS-DOS 4.0. Two new bits are defined, and one bit is no longer used. Unused bits are reserved and should be set to zero. Bit 11, (the OPN, or open bit) indicates whether the device driver accepts I/O request functions in the range 13-15. These are new functions that include Open and Close. Bit 5, (the MS-DOS 4.0 bit) indicates whether the device driver accepts functions in the range 16-18. These functions are described in Section 7.3.

Bit 4, (the SPC, or special bit) was used to indicate that a console output device supported INT 29H as a shortcut for console output. That attribute is no longer supported and console drivers need not provide the function. In MS-DOS 4.0, it is suggested that the console device driver be divided into two independent halves. One of these is completely responsible for output, the other is completely responsible for input. The halves of the console device driver are indicated by the SCR (screen) and KBD (keyboard) bits 1 and 0. In MS-DOS 2.0, these bits were called STI (standard input) and STO (standard output), respectively. The new names more accurately describe the device driver function.

MS-DOS 4.0 ATTRIBUTE WORD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CHR	IOC	IBM	///	OPN	///	///	///	///	///	4.0	SPC	CLK	NUL	SCR	KBD

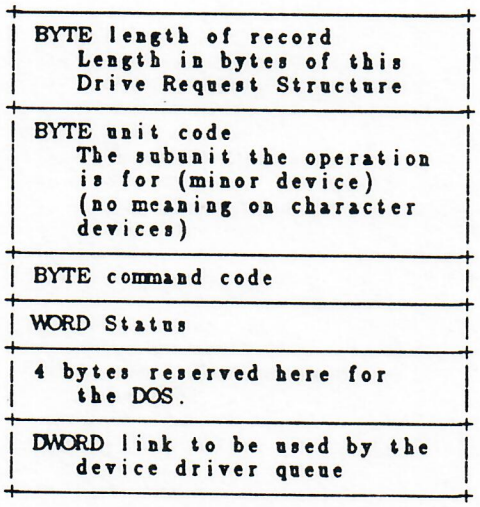
7. I/O REQUEST PACKET

The format of an MS-DOS 4.0 request packet remains the same as in MS-DOS 2.0, except for changes to the request packet header, described below.

7.1. Request Header

There are two changes in the header of the request packet. First, four of the reserved bytes in the header are now provided to the device driver so it can queue the requests internally. The other four bytes remain reserved for future expansion.

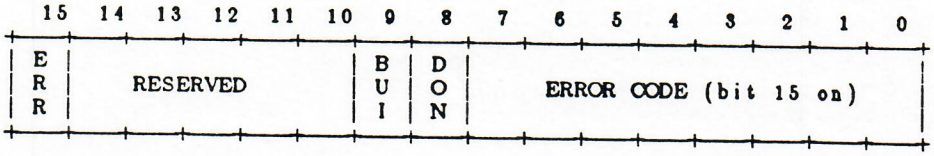
MS-DOS 4.0 REQUEST HEADER



7.2. Status Word

The second change is in the status word. The DONE bit is used to signal if a request has actually been completed. In MS-DOS 2.0 device drivers, the DONE bit was always set upon return to the DOS from the device driver. In MS-DOS 4.0, the driver may return with the DONE bit not set. The DONE bit is normally set by the interrupt service routine that then awakens the corresponding process, using the **DoneRequest** helper.

MS-DOS 4.0 STATUS WORD

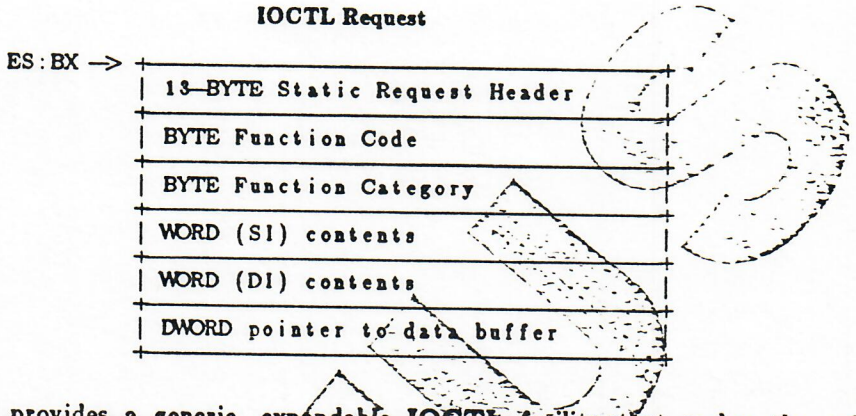


7.3. New Commands

There are three new commands for device drivers. If a device driver does not support any of these new functions it should return error code 3 (Unknown Command). These new MS-DOS functions are:

16 - Generic IOCTL Request
17 - Stop Device Request
18 - Restart Device Request
They are described in the following sections.

7.3.1. Generic IOCTL Request



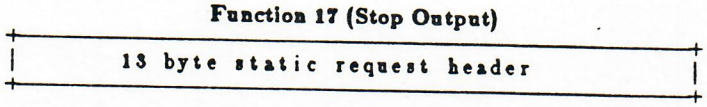
This function provides a generic, expandable **IOCTL** facility that makes the existing **READ IOCTL** and **WRITE IOCTL** device driver functions obsolete. The MS-DOS 2.0 **IOCTL** functions will remain to support existing uses of the **IOCTL** system call (subfunctions 2, 3, 4 and 5), but all new **IOCTL** functions will use the generic MS-DOS 4.0 **IOCTL** facility.

The **IOCTL** facility provides device-specific control functions. Many of these functions are independent of a particular hardware interface card and its associated device driver. For example, the function that specifies which key-codes are to generate asynchronous *signals* (keyboard intercepts) is specific to console keyboard devices, but is not specific to a particular keyboard device.

MS-DOS 4.0 includes many of these device-specific but hardware-independent functions into its own code. This simplifies the task of writing a device driver and ensures consistent behavior among a diverse collection of device drivers. The generic **IOCTL** function contains both a function code and a function category. The DOS examines the category field in order to intercept and obey device commands that are actually serviced by the DOS code; all other command categories are forwarded to the device driver for servicing.

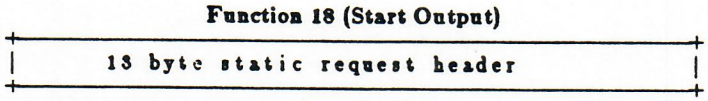
The command categories and the DOS-provided hardware-independent device services are discussed in the following section. One particular category, *screen switching*, is discussed in some length in Section 9, "The Console Device."

7.3.2. Stop Device Request



Function 17, Stop Output, is issued only to console screen drivers. It instructs them to suspend any further output to the currently active console. Any write requests to that console image should suspend (**ProcBlock**) until a Start Output (Function 18) is received.

7.3.3. Restart Device Request



Function 18, Start Output, is issued only to console screen drivers. It instructs them to resume output on the currently active console. See Section 7.3.2.

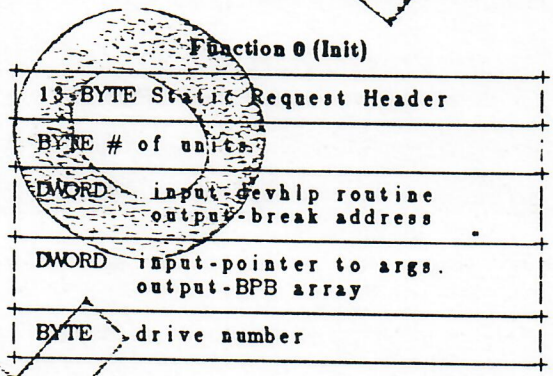
7.4. Changed Requests

The following requests have changed slightly for MS-DOS 4.0:

Command Code	Function
0	Init
2	Build BPB
4	Input (Read)
5	Non-Destructive Input No Wait
6	Input Status
7	Flush Input
8	Output (Write)
9	Output with Verify
10	Output Status
11	Output Flush

7.4.1. Init Request

The Init request now defines two new input-parameters. The devhlp routine is a pointer to the DOS device helper function. See Section 5 and Section 10 for more details. The drive number field gives the first drive number that would be assigned to a block type device.

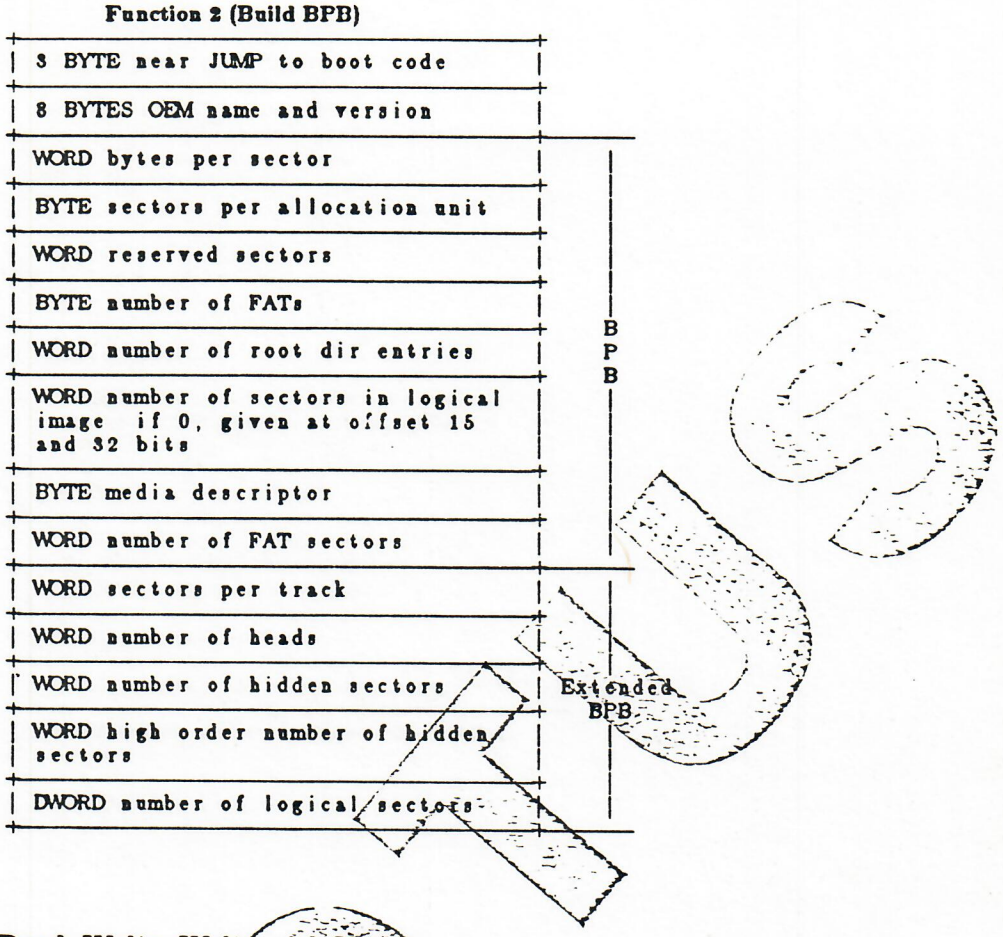


7.4.2. Build BPB (BIOS Parameter Block)

The definition of the BIOS Parameter Block has changed to allow definition of devices with more than 65535 sectors. The BPB will be considered to be in the old format if the word at offset 8 of the BPB is non-zero. The request header is unchanged.

This is the format of the BPB:

Function 2 (Build BPB)

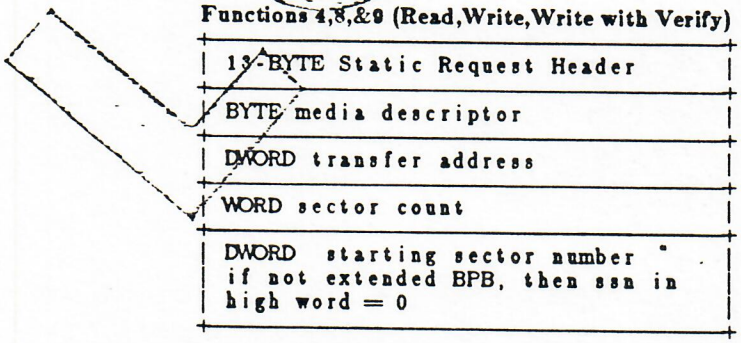


7.4.3. Read, Write, Write with Verify

For block devices:

The starting sector number field has been extended to 32 bits. This will be passed as a 16 bit number unless the extended form of the BPB described in the previous section is used.

Functions 4,8,&9 (Read,Write,Write with Verify)

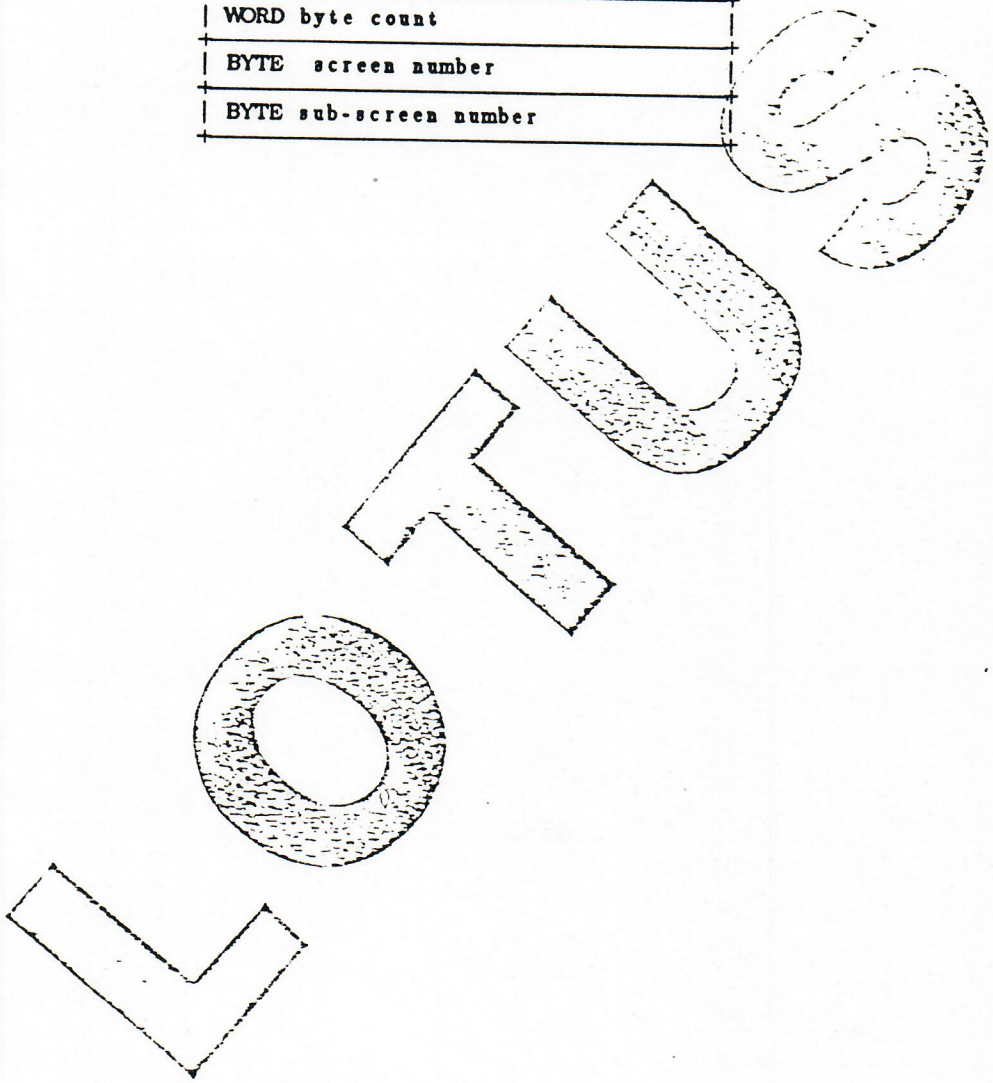


For character devices:

For screen, keyboard, and mouse drivers only, the previously unused field that gives the starting sector number for block devices will now give the screen and sub-screen numbers. See Section 9 for more details.

Functions 4,8,&9 (Read,Write,Write with Verify)

13-BYTE Static Request Header
BYTE media descriptor
DWORD transfer address
WORD byte count
BYTE screen number
BYTE sub-screen number



8. OPERATION

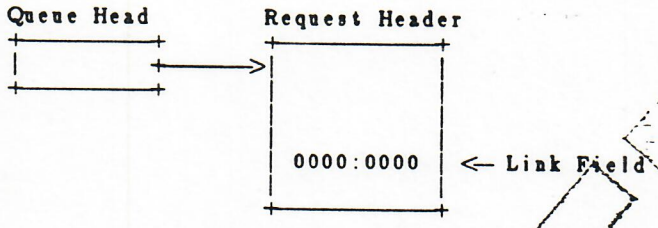
8.1. Internal Queues

Under MS-DOS 4.0 a device driver must either queue the request or carry it out until completion (like an MS-DOS 2.0 driver). Normally only read and write requests are queued; others can be handled immediately.

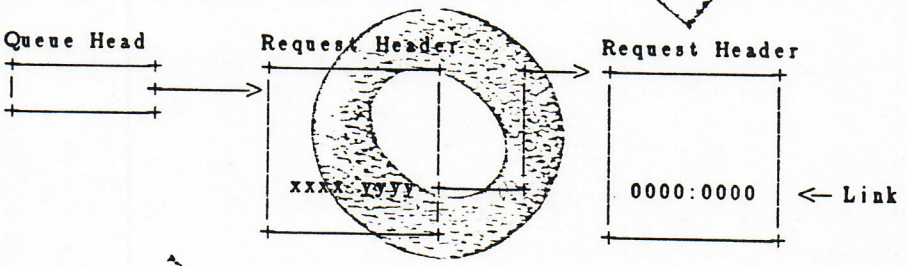
A block (disk) device driver is free to act upon queued requests in any order. The decision to choose queued requests in a particular order is left to the writer of the device driver. Serial (character) device drivers should always handle requests in order, or the output will be mixed.

Once a request is issued to the device driver it is guaranteed that the location in memory of the request packet will not change; thus the device driver may place the location of this request in the link field of the previous request packet. By convention, the end of the list is marked by setting the link field to zero (both segment and offset).

For example, device driver FOO is a block device and uses a request queue. It was inactive until recently called by the DOS for an I/O operation. It has queued the request and started the device. The request queue would look like this:



While still waiting for the I/O interrupt, MS-DOS 4.0 issues a new request. The driver queues the new package by placing the pointer to the new request in the link field of the package in progress. It does not call its internal Start routine because it notes that an operation is already in progress. The queue looks like this:



The queue can be extended as more requests are issued. Note that the **Strategy** routine should never place a new request at the head of a non-empty queue, since the first element in the queue is the active one. Also note that the **Strategy** enqueue code "critical section" must lock out interrupts from the device.

Once the device driver is done with all pending requests, the *HeadOfQueue* will be zero, indicating there are no more requests.

8.2. Request Completed Communication

Every time a device driver completes a request, either successfully or due to an error, it must signal the DOS that the request is done.

The signaling of the DOS is accomplished by a long call that the device driver makes to the DOS with the address of the request completed. The routine being called is the DOS device helper function **DoneRequest** (see Section 5.1, "MS-DOS Supplied Services").

The device driver returns the pointer to the request completed in ES:BX. Note that all the pertinent fields in the request should have been updated before the device driver issues this long call.

Once the DOS returns from the long call, the device driver can process a new request if any are presently in its internal queue; otherwise, it just executes an IRET.

8.3. Nesting Interrupts

Due to many factors, including the amount of free stack space, nested interrupts are not allowed unless special actions are taken. Normally, interrupts are disabled during the entire time the device driver interrupt service routine is active. This means that a high-priority interrupt will not interrupt a low-priority routine already being executed. This mechanism is fast and efficient; most interrupt service routines are done quickly enough that the lockout of higher priority interrupts is not a problem.

Occasionally an interrupt service routine may be slow enough to necessitate the re-enabling of interrupts. The DOS provides a kernel service routine (called via an **INT 32H** instruction) to facilitate this. The interrupt service routine does the following:

- Calls the DOS **allow nested interrupt** helper routine. This routine makes sure the stack is large enough and takes care of other administrative chores.

  Note that the helper routine must be called with all registers (including SP) as they were at the time the interrupt service routine was entered. If the device driver has to do something before calling the helper routine, it must restore *all* registers to their original state before making the call.
- Does an STI after the return from the **INT 32H**. The currently active interrupt may also have to be dismissed to allow other interrupts to be processed.
- Does a normal IRET when done. Control will return to the DOS which will take care of its bookkeeping and do the actual IRET.

Note that unless the **allow nested interrupt** helper is called, any processes that were allowed to run via a **ProcRun** cannot actually run until some other interrupt handler does an **allow nested interrupt** call, a clock interrupt occurs, or the running process executes a system call. For frequently occurring interrupts, the overhead of this call should be avoided if possible.

8.4. Initialization

The format of the INIT request issued to initialize a device driver (at boot time) remains the same, but the field used by the device driver to set the break address when it exits the initialization routine now contains on entry the FAR address of the DOS routine used to access the DOS services. This address should be saved for later use by the device driver. (See Section 5.1, "MS-DOS Supplied Services.")

8.5. Non-Interrupt-Driven Devices

For maximum I/O throughput, interrupt-driven devices should be used. In cases where that is not possible, two techniques may be used to avoid polling in a CPU loop, and to allow the CPU to run other tasks. Note however, that CPU polling will work. This allows MS-DOS 2.0 device drivers to work without modification.

One technique that will work with devices which are part of the resident BIOS is to have the **Interrupt** routine driven off of the system clock's interrupt handler. MS-DOS 4.0 requires an interrupt-driven clock handler in order to drive the scheduler. The clock interrupt handler will become the interrupt-time component of the device driver. It will check the status of the device and call **DoneRequest** and the device's **Start** routine when appropriate.

Another technique is to have the **Strategy** routine call **ProcBlock** with a *time-limit* set. When the *time-limit* expires, the device status could be checked and the interrupt routine called when the device is ready.

9. THE CONSOLE DEVICE

The console device represents a special type of device. Whether the CON device driver is installed or is part of the BIOS, it must conform to the new MS-DOS 4.0 model. It is rare to install a new console device driver in a system; the CON device driver is normally part of the resident BIOS and is not replaced. However, in order to allow the resident BIOS screen and keyboard drivers to be separately and independently overridden, the input and output halves of the console device should be structured as separate drivers. The DOS will recognize the two drivers by the SCR and KBD attribute bits in the device header, and it will direct I/O requests to the proper driver (in some circumstances, an **IOCTL** request may be sent to both drivers).

9.1. Keyboard Interrupts

The DOS must look at incoming keyboard characters to determine if they would send asynchronous signals to some process. The keyboard interrupt handler must pass these characters to the DOS's Key Intercept Handler before the characters are placed in any internal buffer of the device driver. The routine should be called whether or not there is an outstanding input request from some task.

The character is passed to the Key Intercept Handler by placing it in the AX register and making a long call to the DOS **ConsInputFilter** helper function. If character is less than 16 bits, then the high bits should be zero, with no parity.

Upon return from the handler, the character will still be in AX, and the zero flag will be set if the character should *not* be placed in the internal buffer of the device driver (i.e., zero set if the character should be ignored). If zero is reset, then the character should be handled normally by passing it to a waiting task or placing it in an internal buffer.

In addition to passing individual characters, the driver must specifically recognize the character (or character sequence, or other special action) that indicates the special signal to the screen manager. When this event is recognized, the **Signal_SM** helper function should be invoked. If the keyboard handler distinguishes several screen switch characters, the desired screen number should be passed in AL. Otherwise, AL should be set to zero.

9.2. Multiple Screen Images

The "starting sector number" field had meaning only for block devices. Under MS-DOS 4.0 this field retains its meaning for block devices, but holds the keyboard or screen number (0 to n–1), for input or output using the keyboard or screen devices. The number of screens (n) implemented is determined by the driver. The system will never use more than eight different screens. The following figure shows the format of the multiple screen image request.

Multiple Screen Image Request

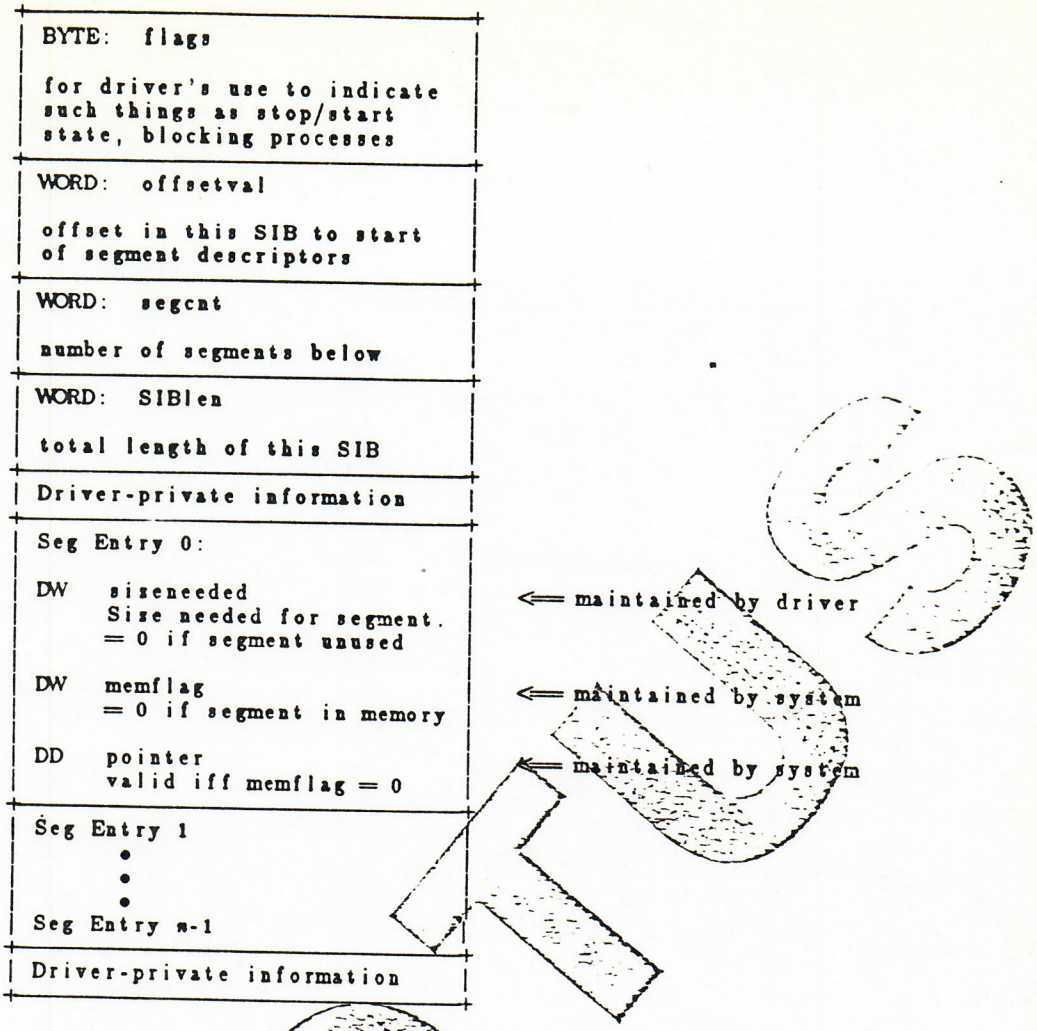
13 byte static request header
BYTE: Media descriptor (unused)
DWORD: Transfer address
WORD: Byte count
WORD: Screen Number

For output, the starting sector number field enables the screen driver to know on which screen it should display this data. If the specified screen image is the currently selected screen, then the device driver should do the output. If the screen image is not the currently selected screen, then the device driver may update the "background screen" by writing into the memory locations where the screen image has been saved. If the driver cannot do so, then it should call the **ProcBlock** helper routine with a code that is unique to this driver and the specified screen image. The address of the associated Screen Information Block (SIB) is suggested as the *ProcBlock-code*. When a non-current screen image is selected for display, the driver should call the **ProcRun** helper function with the same address that was passed to the **ProcBlock** helper. The awakened process can then continue its write to the screen.

For input, the screen number field tells the keyboard driver which "keyboard image" it should use to read the data. Typically, a keyboard driver maintains (with the aid of the **QueueWrite** and **QueueRead** helper functions) a separate queue for each keyboard image; there is a corresponding keyboard image for each screen image. When a key is struck, the keyboard interrupt service routine places the character in the queue belonging to the active keyboard, after having determined that the character should be queued. Read requests are satisfied from the proper keyboard queue; if the queue is empty the driver must call **ProcBlock** until the interrupt service routine adds more characters to the queue. It is suggested that the driver call **ProcBlock** with the address of the appropriate keyboard queue.

Note that little extra code is required to handle the multiple keyboard images because the driver always reads until the request is satisfied or it must call **ProcBlock** awaiting additional keystrokes. Processes belonging to the non-current keyboard/screen group can run until the type-ahead in their keyboard image's type-ahead buffer has been read. When the type-ahead buffer becomes exhausted, the processes must wait until that keyboard image becomes the active one and another key is struck.

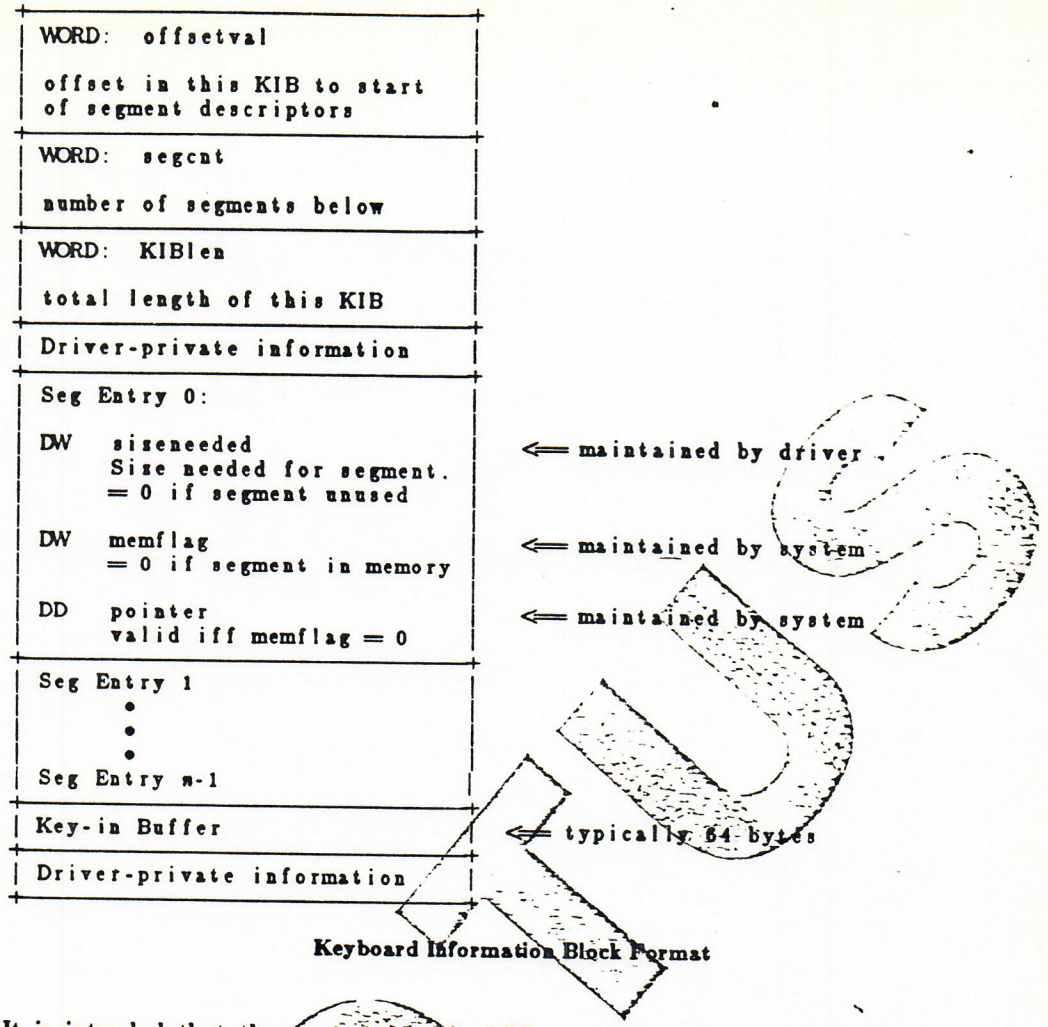
The data that defines the content and state of a particular screen or keyboard image is called a Screen Information Block (SIB) or a Keyboard Information Block (KIB). The data is in a well-defined format local to the device driver and is communicated to the system and knowledgeable system programs via **IOCTL** calls. The SIB format appears on the following page.



Screen Information Block Format

It is expected that the driver will place relatively large, variable-length data (such as the screen image) in the segments. State information about the screen may be placed in a segment or in one of the areas reserved for driver-private information.

The KIB format appears on the following page.



Keyboard Information Block Format

It is intended that the segments in the KIB are to be used for large variable-length data such as keyboard mappings. They could also be used for a type-ahead buffer, but since the data required is relatively small and its immediate availability is useful, it is suggested that the type-ahead buffer be a dedicated part of the structure. In general, it is expected that KIB's would have no segments allocated.

9.3. Screen Switching

The screen manager uses the generic **IOCTL** facility described earlier to communicate screen switching commands to the device driver. One of two categories of **IOCTL** is used, depending on whether the information is exchanged with the screen or keyboard driver. The categories and codes used are:

```
IOC_SC  =       3               ; Screen Control
        IOSC_LS =       41              ; Locate SIB
        IOSC_SS =       42              ; save segment
        IOSC_RS =       43              ; restore segment
        IOSC_EI =       44              ; re-enable I/O
        IOSC_IS =       45              ; initialize screen

IOC_KC  =       4               ; Keyboard Control
        IOKC_LK =       41              ; Locate KIB
        IOKC_SS =       42              ; save segment
        IOKC_RS =       43              ; restore segment
        IOKC_CK =       44              ; change keyboard images
        IOKC_IK =       45              ; initialize keyboard
```

A similar command sequence is issued to the screen and keyboard driver to effect a screen switch. Below is the general sequence for a screen switch. A keyboard switch is similar except that the corresponding calls to **IOCTL** for keyboard control are used.

```
IOSC_LS                         locate old screen SIB
for each segment
    IOSC_SS                     save screen image
IOSC_LS                         locate new screen SIB
if first time
    IOSC_IS                     fill screen with blanks
else
    for each segment
        IOSC_RS                 restore screen image
IOSC_EI
```

From the point of view of the driver, I/O to the device may be suspended between the first **IOSC_LS** (or **IOKC_LK**) and the **IOSC_EI** (or **IOKC_CK**) command. The current screen number should be changed and the corresponding SIB (or KIB) should be selected whenever an **IOSC_LS** (**IOKC_LK**) or **IOSC_IS** (**IOKC_IK**) command is received. The screen number is passed in the User SI field of the request packet.

When an **IOSC_LS** (or **IOKC_LK**) command is received, the driver should initialize the *Sizeneeded* field of each segment. When a save segment command is received, the driver should save the data associated with the segment indexed by SI. Similarly, when a restore segment command is received, the driver should restore the data associated with the indexed segment.

9.4. ROM Emulation

The console driver in MS-DOS 4.0 must emulate ROM routines that handle keyboard input and are conventionally called by applications in the MS-DOS 4.0 environment. (An example is **INT 16** on the IBM-PC and compatibles.) The console driver must do this to allow the system to run other tasks while one task is waiting for keyboard input and to properly direct the input in the multi-screen environment. As a consequence, the console driver should not use the interrupt to get input characters to satisfy read requests, although it may simulate an interrupt to the actual ROM routine to get characters. In addition, other device drivers must not issue the interrupt if they are running in system mode (i.e., in response to an **INT 21** request).

When a user process requests input characters via an emulated ROM routine, there is no screen image number passed along with the request that would allow the driver to determine from which KIB type-ahead buffer to get the characters. The emulating routine should instead use the **GetDOSVar** helper function to find the address of the variable *ScrnIoOk*. This variable will be non-zero when the currently executing process is attached to the currently active screen. If the variable is zero, the emulating routine should call **ProcBlock** on the process until the variable becomes non-zero. The address of the *ScrnIoOk* variable should be used as the *ProcBlock-code*; the DOS issues a **ProcRun** on this code when the currently active screen changes.

9.5. Other Issues

To allow proper, installable versions of the console driver, the driver must honor de-installation requests. Like initialization requests, these will be issued at the user level so that **INT 21** functions may be called. The driver must restore all interrupt vectors it had set and release any dynamic memory it may have allocated. It will return an indication of the memory occupied by its code and data.

The subsequently loaded console device must take over *all* responsibilities of the overloaded console driver, including keyboard filtering and buffering (if it is a keyboard driver), and the management of the screen switch facility.

10. DEVICE HELP FUNCTION DETAILS

The DOS-supplied services listed in Section 5 are accessed by loading a function code into DL and making a FAR call to the routine whose address was supplied at device initialization time. The function codes are defined symbolically in the file **devhlp.inc**. The text of the **devhlp.inc** file follows:

```
SUBTTL  DevHlp - Definitions for device driver helper functions

;       SCCSID = @(#)devhlp.inc 1.2 84/07/16

int_savregs     EQU     32H     ; interrupt routine which saves
                                ; interrupt-time registers.  Nothing except
                                ; the Flags, CS and IP may be on the stack.


DevHlp_SchedClock       EQU     0       ; Called each timer tick
DevHlp_DevDone          EQU     1       ; Device I/O complete
DevHlp_PullRequest      EQU     2       ; Pull next request from Q
DevHlp_PullParticular   EQU     3       ; Pull a specific request
DevHlp_PushRequest      EQU     4       ; Push the request
DevHlp_ConsInputFilter  EQU     5       ; Keyboard intercept check
DevHlp_SortRequest      EQU     6       ; Push request in sorted order
DevHlp_Signal_SM        EQU     7       ; Send signal to session manager
DevHlp_ProcBlock        EQU     9       ; Block on event
DevHlp_ProcRun          EQU     10      ; Unblock process
DevHlp_QueueInit        EQU     11      ; Init/Clear char queue
DevHlp_QueueWrite       EQU     13      ; Put a char in the queue
DevHlp_QueueRead        EQU     14      ; Get a char from the queue
DevHlp_GetDOSVar        EQU     15      ; Return pointer to DOS variable


;       Character Queue structure
;
;       QueueInit must be called before any other queue manipulation
;       subroutine.  The Qsize field must be initialised before
;       calling QueueInit.

CharQueue STRUC
        Qsize   DW      ?               ; Size of queue in bytes
        Qchrout DW      ?               ; Index of next char out
        Qcount  DW      ?               ; Count of characters in the queue
        Qbase   DB      ?               ; Queue buffer
CharQueue ENDS

SUBTTL
```

Calling conventions for each of the helper routines follow. In addition to the explicit effects noted under each routine, the interrupt flag may be set or cleared by some routines, and other flags may be affected by the calls. Some routines require that the interrupt flag be off when they are called.

10.1. Request Queue Management Routines

```
;**      PullRequest — Pull request packet from linked list
;
;       PullRequest pulls the next waiting packet address from the selected
;       device queue.  If there is no packet, then the zero flag is set on
;       return.
;
;       ENTRY   DS:SI           Head of device list (should match PushRequest value).
;
;       EXIT    Zero            Set if there is no request to fulfill.
;               ES:BX           Pointer to device packet.
```

```
;**     PullParticular — Pull particular request packet from list

;       PullParticular pulls the specified packet address from the selected
;       device queue.  If the packet is not found, then the zero flag is set
;       on return.

;       ENTRY   DS:SI       Head of device list (should match PushRequest value)
;               ES:BX       Pointer to device packet.

;       EXIT    Zero        Set if the specified request is not found.


;**     PushRequest — Push request packet on linked list.

;       PushRequest adds the current device request packet to the list of
;       packets to be executed by the device interrupt routine.  When the
;       device routine finds that the request can not be immediately done,
;       it calls this routine that adds the packet to a list.  Since the
;       device is active at this point, the caller must have turned off
;       interrupts before checking to see if the device was busy (otherwise
;       a window exists in which the device finishes before the packet is
;       put on the list).

;       ENTRY   DS:SI       Pointer to DWORD head of device list (next request
;                           to do).  It should be initialised to 0.
;               ES:BX       Pointer to device request packet.


;**     SortRequest — Insert request packet on linked list in sorted order.

;       SortRequest inserts the current device request packet into the list
;       of packets to be executed by the device interrupt routine.  This
;       function is similar to PushRequest except the request packet is
;       inserted in sorted order by starting record number.

;       ENTRY   DS:SI       Pointer to DWORD head of device list (next request
;                           to do).  It should be initialised to 0.
;               ES:BX       Pointer to device request packet.


;**     DevDone — Flag I/O Complete.

;       DevDone is called from the device interrupt routine.  It
;       reschedules the process that owns the request to run again.

;       ENTRY   ES:BX       Pointer to I/O request block.
```

10.2. Process Synchronization Routines

```
;**     ProcBlock — Block this process from running

;       ProcBlock ''sleeps'' (suspends) the current process until some other process
;       (or an interrupt service routine) issues an appropriate ProcRun
;       call.  Both the ProcBlock and ProcRun routines are supplied
;       a 32-bit ''event identifier.''  When ProcRun is called with a
;       particular event identifier it reactivates all blocked
;       processes that also presented that identifier.

;       By convention, the 32-bit ''event identifier'' is the address of
;       some structure or memory cell. For example, processes waiting
;       for I/O to be completed call ProcBlock with the address of the
;       I/O block.  When the device driver completes the I/O and marks
;       the I/O block done it calls ProcRun with that address
;       so that any waiting processes can resume operation.  Most
;       device drivers are interrupt-driven, so ProcRun is often called
;       at interrupt-time by the device interrupt handler.
```

```
When calling ProcBlock it is important to use the sequence:

        Disable Interrupts
        while (need to wait)
                ProcBlock(value)

Interrupts are turned off before checking the condition
(I/O done, resource freed, whatever) first to avoid a
deadlock by getting an interrupt-time ProcRun call before
completing the call to ProcBlock.  ProcBlock re-enables the
interrupts.  Also note the ''while'' clause: it is good practice
to re-check the awaited condition and, if necessary, re-disable
interrupts and re-call ProcBlock.  The convention of using
an address as an 'event identifier' should prevent double use
of an identifier.

ProcBlock does its work by entering the specifics in the
wait queues and then calling the scheduler to run another process.
Naturally, ProcBlock cannot be called at interrupt time.

When the wakeup is issued, or the timeout
expires, or when the task is signaled,
ProcBlock exits to the caller.

ENTRY   AX:BX       Event identifier.
        CX          Timeout interval (in MS)   0 if to never timeout,
                    DH != 0     if sleep is interruptible.
        Interrupts DISABLED to prevent ProcBlock/ProcRun races.
EXIT    'C'         Clear if event wakeup.
        'C'         Set if unusual wakeup.
          'Z'       Set if timeout wakeup.
          'Z'       Clear if process was interrupted.
        AL          Awake code, non-zero if unusual wakeup.
        Interrupts enabled.
USES    AX, BX, CX, DX, FLAGS

EFFECTS SWITCHES CONTEXT

;**     ProcRun — Release Blocked Processes

ProcRun is called to 'wakeup' blocked processes.  All
processes sleeping on the particular event identifier
are awakened, be it 0 or many.

ProcRun is often called at interrupt time.
See ProcBlock for a detailed discussion.

Note that as a convenience, the ''wakeup cause'' was set by
ProcBlock to be PBR_NA (normal awakening) so that we don't
have to set that value in the wakening PTDA.

ENTRY   AX:BX       Event identifier.
EXIT    AX          Count of those awakened.
        'Z'         Set according to AX.
USES    AX, BX, CX, DX, FLAGS
```

10.3. Special Routines for Console and Clock Drivers

```
;**     Cons_Input_Filter — Filter console input characters.

Checks the console input stream for characters that need
immediate treatment.

ENTRY   AX          Input character ASCII code.
EXIT    ZF          Clear if key should be queued.
        ZF          Set if key was eaten.
USES    FL

CALLED FROM         DEVHLP: DeviceHelp
```

```
;**     Signal_SM — Signal Session Manager

;       Signal_SM sends a signal to the Session Manager that may respond
;       to the user's request.  The caller may specify a screen that
;       should be switched to.  If the caller has no particular
;       preference, screen zero should be specified.

;       ENTRY   AL          Desired screen number.
;       EXIT    CF          Set if Error sending signal to SM.
;       USES    AL,FL

;       CALLED FROM         DEVHLP: DeviceHelp


;**     SchedClock — Receive Clock Tic

;       SchedClock is entered from the clock routine every scheduler tick
;       (If clock is very fast, might make the scheduler tick be so many clock
;       ticks to cut down scheduler overhead).

;       The caller supplies the clock ''tick interval,'' in milliseconds,
;       rounded down.
;       For example, a 60 hz clock interrupts every 16.66 microseconds, so
;       the tic interval would be 16.  This value is used by the DOS to
;       compute the number of ticks to count for a particular interval.
;       The tic interval value must not change once the system is
;       ''up and running.''

;       WARNING:  Depending upon the particular BIOS, SchedClock
;                 may be called before the scheduler has been set up.

;       ENTRY   AL          Tick interval, in milliseconds.
;       EXIT    NONE
;       USES    ALL
```

10.4. Character Queuing Routines

```
;**     QueueFlush — Flush characters in queue.

;       QueueInit and QueueFlush are functionally equivalent.  They
;       initialize the queue structure variables to indicate the buffer
;       is empty.

;       ENTRY   DS:BX       Points to the queue structure to be initialized.
;               (The Qsize field must be set up.)


;**     QueueWrite — Put character into queue structure.

;       QueueWrite is called to insert a character onto the end of the
;       specified queue.  If the queue is full, the zero flag is set.

;       ENTRY   DS:BX       Points to the queue structure.
;               AL          The character to insert at the end of the queue.
;       EXIT    Z set       Queue is full.
;               Z clear     Character stored successfully.
;       USES    Flags
```

```
;**     QueueRead — Read a character from a queue.
;
;       QueueRead is called to remove a character from the beginning of the
;       specified queue.  If the queue is empty, the zero flag is set.
;
;       ENTRY   DS:BX       Points to the queue structure.
;
;       EXIT    Zero        Set if the queue is empty, otherwise cleared.
;               AL          The character read from the queue.
;       USES    Flags
```

10.5. Miscellaneous Routines

```
        ReferGlobal         ScrnIoOk,BYTE

dosvartab:
        DW      OFFSET DosGroup:ScrnIoOk,1          ; 0
dosvarend:

        NDOSVAR = ((dosvarend - dosvartab) SHR 2)


;**     GetDOSVar — Get address of important DOS variables
;
;       GetDOSVar will return the address of important DOS
;       variables.  The list of variables available is subject
;       to change with different versions of the system.
;
;       ENTRY   AL          Index of variable wanted.
;               BX          If AL names an array, index into the array desired.
;               CX          Expected length of variable.
;       EXIT    if CF clear,
;                           DX:AX is the address of the variable.
;               else something is wrong, any of:
;                           AL out of range.
;                           BX out of range.
;                           CX does not match actual size.
;       USES    AX, BX, CX, DX
```