CHAPTER 3: MEMORY ORGANIZATION

Introduction

Using MOS with the appropriate memory management device driver (MEMDEV) will produce a more flexible system than is available without memory management. This is because of MOS's ability to use extended memory for system and task memory.

MOS loads differently when extended memory is available. The kernel attempts to relocate itself to the memory region from C0000 to F0000 thereby allowing larger amounts of memory below the 640K to be devoted to DOS compatible applications.

When tasks are activated on systems with extended memory and paging capable memory management support, the memory for each task is allocated from extended memory. Otherwise, task memory is allocated from the task that creates each new task. The latter method obviously limits memory available to DOS-compatible applications.

When tasks are created with or without extended memory present in the system, context save areas and video areas are allocated as well as the task applications area.

Context Save Area

The context save area is a block of memory allocated for an individual task's interrupt vector table, BIOS RAM data, a miscellaneous data area, and type ahead buffer. The miscellaneous data area handles print screen status, single drive mode status for systems with only one floppy drive, and diskette initialization.

On boot-up, the context save area is initiated and a master copy is then stored in the System Memory Pool (SMP). When a task is created, a context save area is allocated below that task's application area and the master copy of the context save area is copied to it. Individual context save areas are then maintained for each task.

Video Areas

Addresses from A0000 to BFFFF are reserved for swapping video RAM for each task as it becomes active. This is the I/O buffer for video. EGA video is reserved from A0000 to AFFFF. 4K is reserved for monochrome video beginning at B0000. 16K is reserved for CGA video beginning at B8000. The appropriate amount of memory is allocated with each task for swapping into the video area. When a task becomes active, this memory is swapped into the video RAM area.

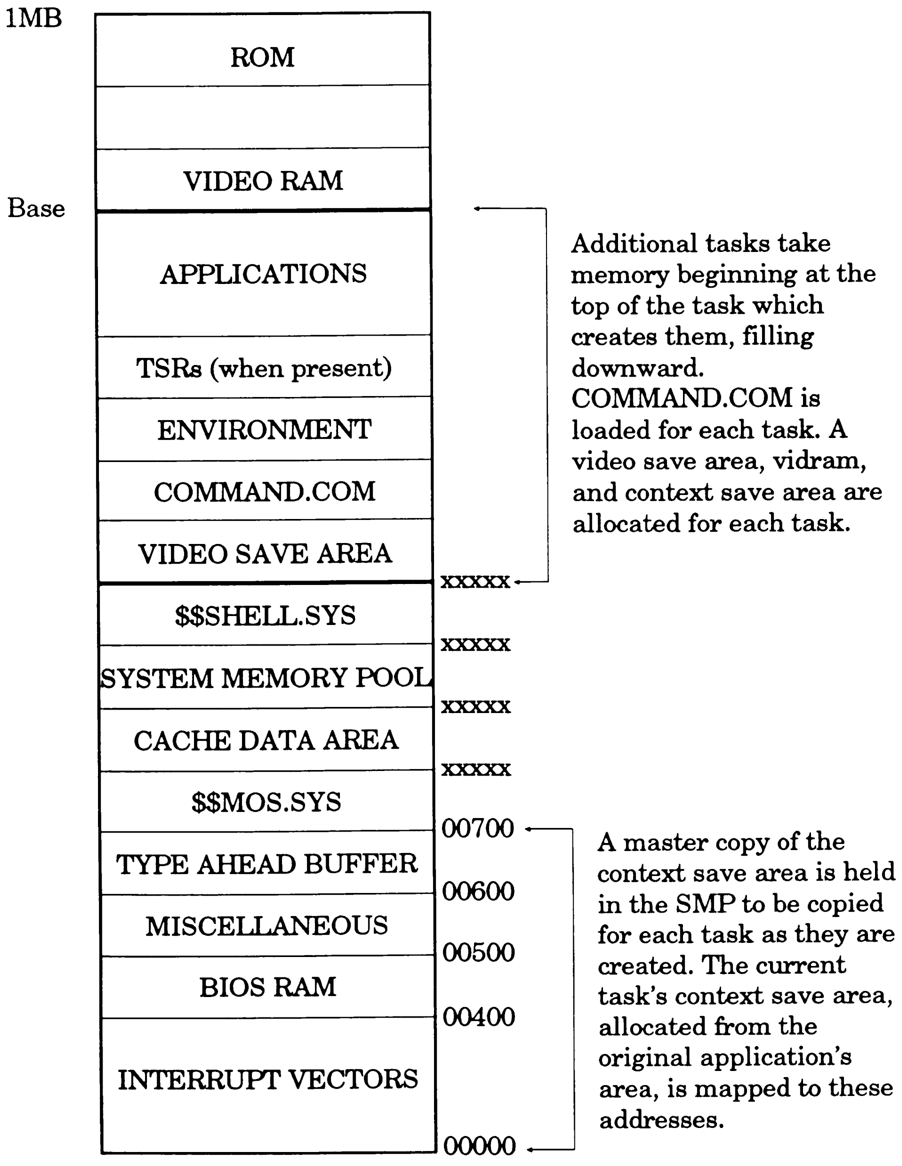
MOS actually reserves two video buffers for each task. In addition to the video RAM area, a video save area is also reserved. For tasks not associated with a dumb terminal only the video RAM area is reserved. The actual addresses used for these buffers are dependent on the availability of extended memory and the memory map for the system.

MOS Memory Map without Memory Management

MOS loads into the lowest available memory beginning at absolute address 00700. The 0 task, or main task, is allocated the remaining contiguous memory for applications. Any additional tasks will take memory from the task that creates them.

Each task will load its own copy of COMMAND.COM and environment; COMMAND.COM requires about 8K and serves as an entry point to $$SHELL.SYS for user or application interface. Additional space is allocated within each task for the video save area. The context save area is allocated from the SMP.

The master copy of the context save area held in the SMP and the amount of memory allocated with each task for the context save area are approximately 1.3K each in systems without memory management.

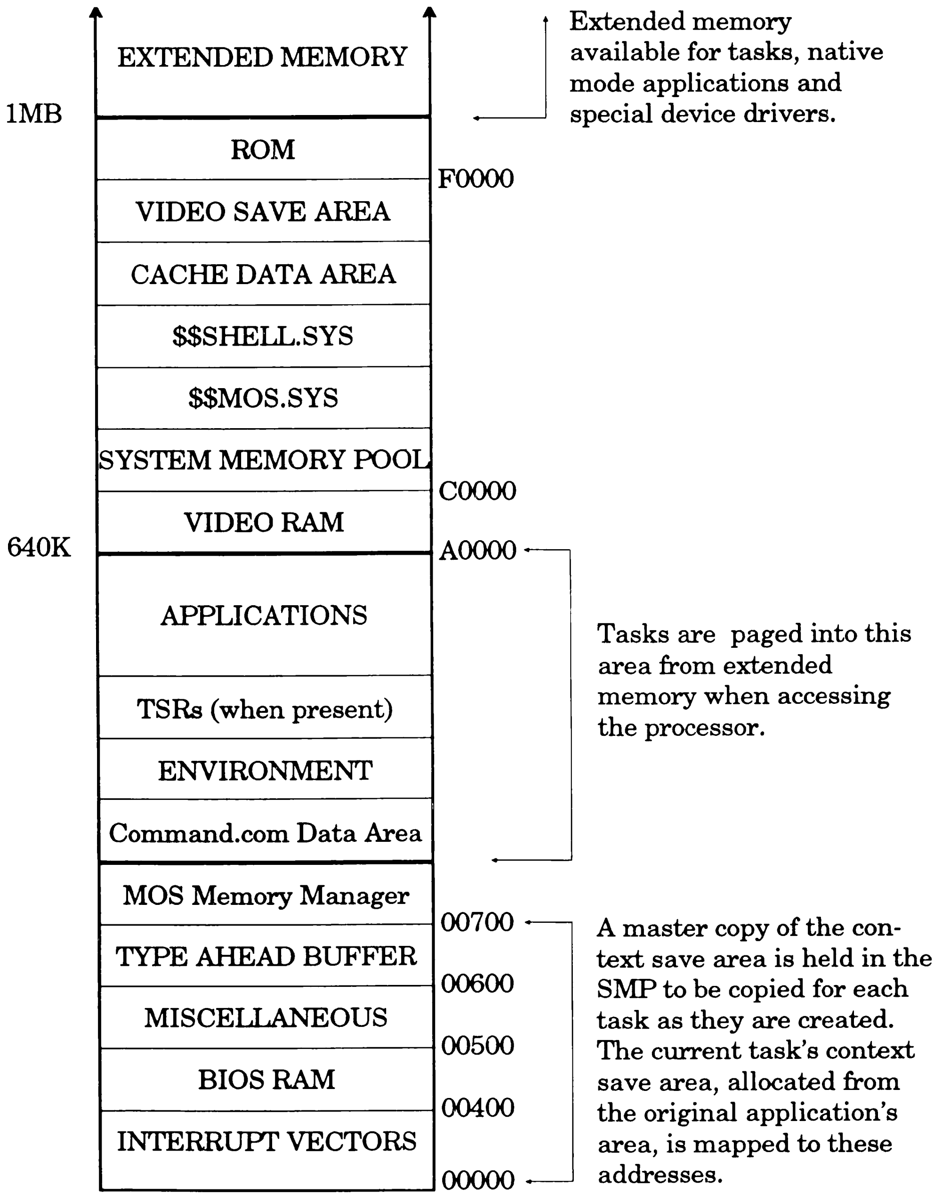

1MB
ROM
VIDEO RAM
Base
APPLICATIONS
TSRs (when present)
ENVIRONMENT
COMMAND.COM
VIDEO SAVE AREA
$$SHELL.SYS
SYSTEM MEMORY POOL
CACHE DATA AREA
$$MOS.SYS
TYPE AHEAD BUFFER
MISCELLANEOUS
BIOS RAM
INTERRUPT VECTORS
xxxxx
xxxxx
xxxxx
xxxxx
00700
00600
00500
00400
00000
Additional tasks take memory beginning at the top of the task which creates them, filling downward. COMMAND.COM is loaded for each task. A video save area, vidram, and context save area are allocated for each task.
A master copy of the context save area is held in the SMP to be copied for each task as they are created. The current task's context save area, allocated from the original application's area, is mapped to these addresses.

Memory map when no memory management is available.

Memory Map in Systems With Memory Management

In systems with the proper memory management, the MOS kernel will typically be able to relocate to higher memory along with the SMP, $$SHELL.SYS and the data areas for the Disk Cache. In addition, the addresses reserved for video save areas will be reserved from higher memory when possible. When insufficient high memory is available for all of these items, as many of them as can fit are relocated up high. The largest items are located first to help insure that whichever items may not fit will be the smaller ones.

MOS dynamically uses the memory management features associated with the memory management device. In 80386 systems, the processor chip has memory "paging" capabilities; other systems re-map memory when appropriate. Extended memory is mapped or paged into addresses associated with the first 1MB for system use as appropriate on boot-up. The remaining extended memory is used for tasks, special device drivers and applications that establish themselves in native mode.


EXTENDED MEMORY
1MB
ROM
F0000
VIDEO SAVE AREA
CACHE DATA AREA
$$SHELL.SYS
$$MOS.SYS
SYSTEM MEMORY POOL
C0000
VIDEO RAM
640K
A0000
APPLICATIONS
TSRs (when present)
ENVIRONMENT
Command.com Data Area
MOS Memory Manager
00700
TYPE AHEAD BUFFER
00600
MISCELLANEOUS
00500
BIOS RAM
00400
INTERRUPT VECTORS
00000
Extended memory available for tasks, native mode applications and special device drivers.
Tasks are paged into this area from extended memory when accessing the processor.
A master copy of the context save area is held in the SMP to be copied for each task as they are created. The current task's context save area, allocated from the original application's area, is mapped to these addresses.

Memory map when memory management is available.

MOS attempts to relocate in the area from C0000 up to but not including F0000; the available location is dependent on the hardware and device drivers in your system. Any EMS emulation software using this range of memory will interfere with the system unless FREEMEM is specified in CONFIG.SYS to force MOS to avoid these areas when relocating. When FREEMEM is absent from CONFIG.SYS, MOS will search higher memory for up to five areas of available memory. The ideal scenario is illustrated above: there is high memory available for relocation of the kernel, SMP, and file buffers. In addition, an area is reserved for mapping of video buffers.

Note that no physical memory is required to be installed within this FREEMEM memory range. MOS uses its memory management features to remap extended memory into the FREEMEM areas which it uses to hold parts of its operating overhead.

If the available memory is fragmented, MOS relocation will be dependent on the size of the memory fragments on a first-fit basis. If the first free memory location is too small for $$MOS.SYS, the following segments will be searched until a large enough segment is located. If no segment located is large enough for relocating $$MOS.SYS, that entity will relocate into lower memory and the operating system will attempt to locate file buffers into the first free memory location.

Beginning with version 4.10, a split SMP feature is available to further optimize the memory allocation process. For example, if a total of 100K of SMP is required but the largest contiguous block of available FREEMEM is 80K, the following statement may be used within CONFIG.SYS:

```
SMPSIZE=80,20K
```

The above will result in two separate SMP blocks, an 80K block and a 20K block. This presumes, of course, that no single device driver requires more than 80K of memory.

DOS Compatibility Considerations

For compatibility with existing products, MOS provides an expanded memory driver to meet with the Lotus/Intel/Microsoft Specification. This driver provides an emulation of an EMS memory adapter by using the remapping capabilities of the memory management driver.

Support is also provided for Interrupt 21H memory functions 48H through 4AH.

Native Mode Considerations

Native mode applications allocate extended memory through MOS's Extended Services Interrupt. After switching to native mode via Function 10H, a program may allocate memory by executing Function 11H.

This memory is established for data and may be designated as stack or code per the program's needs by issuing a Function 13H. Upon termination, the program must de-allocate any such memory by issuing a call to Function 12H, or the memory will remain inaccessible until rebooting the system.

Note that the pushf/call far calling method described in Chapters 8 and 13 must not be used when making calls for native mode services. In these cases, a software interrupt instruction must be executed.

CHAPTER 4: PROCESSOR DIFFERENCES

Differences between 80386 and Non-386 Systems

MOS supports a wide variety of hardware systems. They can be separated into three general categories:

1. No memory management

2. Non-386 memory management hardware (e.g., with the AT-GIZMO)

3. 80386 systems

The most obvious difference between the second and third categories is the ability to run 80386 native mode applications, with multi-megabyte linear addressing capabilities, 32-bit operations, and a sophisticated instruction set.

There are other less obvious differences. In general, the 80386 provides more tools for solving compatibility issues with already existing applications. These are primarily memory protection and I/O protection. They allow us to:

1. Protect the MOS kernel and other programs from destruction by errant software

2. Intercept direct access to I/O ports by applications and allow appropriate emulation for the task

3. Improve performance by hardware detection of direct writes to video memory, thereby avoiding needless repetitive testing for new video data

4. Avoid patching ill-behaved applications to achieve compatibility

One disadvantage of the 80386 is that all interrupts involve considerable overhead, because of the unavoidable transition to native mode at privilege level zero. This adds up to around 200-400 (or more) CPU cycles per interrupt. Therefore, it is wise to avoid frequent "INT nn" instructions in 8086-level applications. A reasonable substitute would be:

```
PUSHF
CLI
CALL   DWORD PTR [vector-contents]
```

This avoids the transition to and from native mode. Do NOT use this technique with native-mode applications.

CHAPTER 5: INTERFACING APPLICATIONS

Introduction

MOS provides two modes of operation on a 80386-based computer, the virtual 8086 mode and the 32-bit native mode. These "software" modes correspond directly to 80386 hardware operating modes of the same name. Procedures for requesting services from MOS for 16-bit applications and 32-bit applications are discussed in this chapter.

16-Bit Applications

When MOS is operating on an 80386 based system, whether it be in real mode (because $386.SYS is not installed) or in virtual 8086 mode, the task switching logic saves the high half of the 32 bit registers. The FS and GS segments registers are also preserved from task to task.

This makes it possible for 16 bit applications running under MOS to take advantage of certain of the 386's advanced features. Note that while data and address size prefixes can be used within virtual 8086 mode to perform 32 bit operations, attempting to address memory above the first megabyte will cause a CPU exception. The native mode environment must be used for this large address model.

32-Bit Applications

When initially loading a program, MOS does not distinguish between 8086 applications and native mode applications, i.e. all programs start in virtual 8086 mode.

To get into native mode, the application must set up a NCA (Native Context Area) block, point ES:BX to it, load 10H into the AH register and issue an call to MOS's Extended Services. It is not necessary for the application to set up 80386 descriptor tables. MOS does that automatically. The Mode Change function (10H) causes translation of the 8086 segment register addresses to protected mode selectors, while building the appropriate descriptor tables. CS is converted to a Code Segment descriptor. SS, DS, ES, FS, and GS are converted to Data Segment descriptors.

Once in native mode, your program may use 32-bit operands and allocate extended memory for code, data, and stack space. The program should de-allocate memory when not needed. It should use "handle" function calls for all I/O, whether file or device related.

CHAPTER 6: THE MULTI-TASKING ENVIRONMENT

Introduction

Programming for a multi-tasking/multi-user environment involves a few more considerations than the traditional single-user environment. With multiple users comes the necessity to protect files and the information in them, yet allow these users to access the data simultaneously. Providing security to sensitive information may be involved. Other considerations include the best way to do I/O, how to set interrupt vectors, and how to go resident in memory.

This chapter is intended to provide suggestions, information, and guidelines for programming in our multi-tasking/ multi-user system. The topics discussed in this chapter are: resource sharing, I/O programming, efficient console input, interrupt handling methods, intertask communication, TSR programs and some general product design considerations.

When designing applications to execute within a MOS enviroment, extra attention must be given to certain characteristics of a multitasking multiuser environment. Since your program will be time sliced it may be running at the "same time" as another copy of your same application or along side any number of other programs. Multiuser programming concerns come in two basic levels - file sharing and machine sharing. Each topic is covered in the sections that follow.

File Sharing

Multi-user applications written for MOS must take advantage of MOS's file and record locking functions. This is necessary to preserve data integrity. It should be obvious that if two or more people are accessing records within a file that some mechanism be used to arbitrate when a task may update a portion of the file and when it may not. The functions to do this are documented in Chapter 8 but we will discuss them a bit further here.

NOTE: All file sharing is provided through the "handle" function calls. Programs that use FCB file I/O will find themselves at a severe disadvantage since file protection is not supported through FCB calls.

To regulate file access, use function 3DH of INT 21H to open a file. The function gives you the ability to set access rights for subsequent file opens by other tasks. For example, the first task to open a file may give itself read/write access to the file while denying write access to the file by all subsequent tasks. Or, the first open call may grant full read/write access to everyone.

File sharing rights are released when the task closes the file. For example, if tasks 1 and 2 both open the same file allowing full access by other open calls and then task 3 attempts to open the file giving itself full access yet denying access for other opens, this final open attempt will fail. The more restrictive open which task 3 is attempting will only succeed when both tasks 1 and 2 close the file.

Record locking provides more precise data protection than file locking in that it protects the records of a file while allowing simultaneous read/write access to the rest of the file. A prime example of this is in financial applications software. Two or more data entry personnel could edit records in an accounts receivable file or an accounts payable file while other users of the program read other parts of the same file since only the records being used would be locked.

MOS's record lock function is 5CH. It allows an application to tell MOS to lock a specified range of bytes in a file. MOS allocates a Record Lock Block (RLB) for the call and associates the RLB with the file. Additional lock calls for the same region of the file cause MOS to scan all RLB's for all tasks associated with the file to determine if the region is "locked". If so, the lock call is retried a set number of times.

Should the retries fail, the lock call will fail. When the application releases the lock, the RLB is removed and its memory space is freed for re-allocation. RLBs are also checked during read and write calls.

These file and record locking mechanisms are suported intrinsically within the MOS kernel. There is no need to load a SHARE.COM TSR. If an application designed for a DOS-based network makes an INT 2F call to see if SHARE has been loaded, MOS's INT 2F handler will return an indication that it has been, even though there is technically nothing seperate to load in. This emulation is done for the benefit of well- behaved network programs which verify the existence of SHARE.

Machine Sharing

File sharing cosiderations can apply when the applications involved are running within two different machines which are connected through a network. They can also apply in the case where two or more applications are running within the same computer system under the control of a multitasking enviroment. Another type of resource sharing is machine sharing. This design consideration only applies in the case where multitasking is being done within one computer.

Designing programs with an awareness of machine sharing involves the skills of "defensive" and "clean" programming. Clean programmers are careful to leave things in the same condition as they found them. Defensive programming tactics can be necessary to maintain stability when sharing a machine with programs that don't clean up after themselves. There are also situations where a defensive approach is required to prevent a task switch from occuring during a sensitive operation.

To understand what's involved in this extra level of awareness and why it is necessary, consider the difference between a single application running under PC-DOS™ and multiple applications running under PC-MOS. In the single-tasking case, the application can assume almost total control of the machine. It can re-program systems hardware in almost any manner which suits its purpose without concern for possible effects on any other processes, since none exist. This is often done because it is the quickest and simplest way to achieve certain results.

When a time slicing multitasking operating system is brought into the picture, the program design philosophy of total machine ownership is basically not a healthy one. The 80386 processor, with its I/O trapping capabilities, can do much towards taming bad behavior. However, sometimes the only practical recourse the I/O trapping logic can take is to deny an operation rather than provide any emulation. When this occurs, or when MOS is running in a system where I/O trapping is not supported, such as an 80286 based machine, designing programs that leave thing in the save condition as they were found is a must to prevent conflicts.

When programs are designed with little or no awareness of the characteristics of a time slicing environment, a number of reliability problems can arise. Something as simple as using the spare channel of the 8253 timer chip to generate sound can cause a full machine lockup when proper programming practices are not followed. An example of this will be illustrated in the next section on I/O Programming.

I/O Programming

There are many instances in which resource sharing concerns are warranted. One common problem area arises when an application is interacting with some of the standard hardware present within the computer, or with external hardware devices which are installed. I/O operations involving multiple step sequences must be given especially careful consideration.

Multi-step I/O sequences are often required when programming hardware such as the 8253 timer/counter chip, video display controller chips, UART's, DMA controllers, etc. Since MOS presents a preemptive time slicing environment, any time that an IRQ0 can occur, the currently executing process may be suspended and another task given its share of the CPU's attention. IRQ0 is another name for an INT 8 -the timer tick interrupt.

Therefore, when performing a multi-step I/O sequence it is very important to disable interrupts. This, of course, must be done only for the shortest time possible so that IRQ's are not lost.

The consequence of not taking this approach is that the order of your I/O sequence may be corrupted. If the first step of a two-step operation is required to select the data register which will be accessed in the next operation, not gaurding against a task switch could result in a scenario such as illustrated by the time sequence list shown below. This sensitivity can also exist when two individual bytes must be written to supply a word value to a device.

The normal sequence that was intended:

1. Task A programs device X for access to register T by writing a control code to its index register.

2. Task A writes a new value to register T of device X.

What could happen in the worst case:

1. Task A programs device X for access to register T by writing a control code to its index register.

2. An IRQ0 occurs which causes task A to be suspended in favor of task B.

3. Task B programs device X for access to register U.

4. Task B reads register U.

5. Another task switch occurs and the exection of Task A is resumed at the point where it was interrupted. However, the write operation that Task A is about to perform will be directed to register U rather than register T!

A more concrete example of this type of problem involves an application which uses the spare channel of the 8253 timer/counter chip to produce sound or to measure time. The application program in which this problem was intially encountered was doing something which seemed fairly innocent.

When an invalid choice was made at a menu, a tone was produced in the system's speaker - a beep. However, the 8253 requires a two-step programming operation and this application was not disabling interrupts around the I/O sequence. In addition, the disk logic within the ROM BIOS of some computer systems reads the first channel of the 8253 to measure short periods of time (for disk head settling time, etc.).

The task setting up the beep was able to get switched out from a time slice interrupt after having only done the first of the two I/O operations to the timer chip. When another task on the system performed a disk access while the timer was left in this intermediate state, the 8253 became totaly confused and stopped generating IRQ0's. A power-down reboot was the only recourse at this point.

When MOS is running in a system where the I/O trapping capability of the 80386 is available, applications which fail to protect themselves from being sliced during access to the 8253 are detected and managed to prevent the timer chip from getting corrupted. MOS virtualizes the device access by queuing up the first I/O operation and waiting until after the second operation before actually accessing the hardware.

However, in systems without I/O trapping capability (e.g. 80286 and 8088 based systems), the only solutions available are to not use applications which present this problem or have them rewritten or patched. The defensive approach here is to examine each portion of your application's code and ask yourself "what would happen if a slice happened here?".

Efficient Console Input

One of two basic program design methods may be employed when requesting console input. In the first approach an operating system call is used which doesn't return until a keystroke is obtained. A call to function 0 of INT 16H is a prime example of this. This method can be used when the only concern is with getting the next keystroke. Under MOS, INT 16H calls are handled by the kernel rather than the ROM BIOS.

A second method which may be used is to make a system call such as INT 16H function 1. This service checks for an available keystoke without waiting until one is available. It returns immediately to the caller with an indication of keyboard status.

This polling type of console input is especially useful when the application must handle multiple processes with a measure of concurrency. For example, some applications maintain an on-screen clock display or some other real-time data while waiting for console input. A word processor which supports background printing is another example as is an applcation which polls both the keyboard and a mouse.

When a program is in a mode where it is waiting for the next keystroke from the user, having the task be suspended while that wait condition exists is highly desirable. The use of an operating system function which waits until a key is pressed affords the greatest efficiency. If no keystroke is queued in the type ahead buffer, the kernel can simply suspend the task until a key is pressed. There is no sense in wasting CPU time on a task which doesn't need it.

Consider for a moment what happens in a multitasking system which is supporting two tasks. Suppose that one task is truly busy sorting database records, and the other is busy only because it is using a polling form of console I/O to wait for a keystroke. For simplicity, presume that both tasks have their time slice count set to one and are of equal priority.

In this situation, each task will execute for an average of 1 time slice out of every 2. However, if the task polling the keyboard could be suspended until a key is pressed, the other task could receive the CPU's attention for 2 out of every 2 time slices. This results in a doubling of the throughput of this "truly busy" task during the time when the other task is waiting for input.

Automatic Task Suspension

The PC-MOS operating system provides an optional monitoring feature which can detect when a program is operating in a keyboard looping mode. With this option, when an application is "hogging" the CPU while polling for a keystroke, it will be suspended if it loops over 8 times in a row without finding a key or without making certain additional system calls.

Once a task has been suspended because it has looped more than 8 consecutive times, it will be re-activated immediately if a keystroke is available. A task suspended in this way is also re-activated once every 18 timer ticks in order to give it a small amount of CPU time -- so that any other processes it must maintain are not completely stalled.

This monitoring condition is inactive by default. It may be enabled through the use of certain utility programs supplied with MOS or through a call to one of MOS's Extended Services function calls. The MOS DIS command can be used to enable this feature from the command line. MOS NODIS is available to return the system to its default state.

To make things more convenient, the TSR pop up windows available with the SPOOL.COM and MONITOR.COM programs can be used at virtually any point to toggle this feature on and off. The alternative of making an Extended Services system call is, of course, only available to applications which have been designed to interface with MOS's API functions.

Some applications loop for console input when they actually don't need to and can be used successfully with MOS's automatic disable feature. However, certain application programs will have problems when this feature is enabled. These are programs which need to loop for a specific reason -- so they can attend to more than one function at the same time.

Such compatiblity issues require that the user be able to interpret conditions and make judgements based on the internal workings of an application. Needless to say, this is a lot to expect. Whenever possible, it is always better to design applications with an awareness of "CPU hogging" and prevent it in the first place.

Often times, when an application is making repeated calls to function 1 of INT 16H, it is making other system calls as well. Note that the application does not have to be making the INT 16H calls directly. Calling a system service such as function 0BH of INT 21H will eventually result in a call to INT 16H function 1.

When MOS has been placed in DIS mode, it is keeping tabs on which system calls are made in conjunction with keyboard status checks. For any calls which indicate that the task is not idle, such as disk I/O, writing console output, or printing, the internal DIS mode counter is reset.

However, if the only other system calls which are made while repeated keyboard status checks are being made are functions 2AH or 2CH of INT 21H, it is presumed that the task is actually idle and is simply using the time/date information to maintain an on-screen clock. Therefore, one consequence of using DIS mode with a program which maintains a clock display while polling for keyboard input is that the clock display may not always stay current. If this is a problem, do not use DIS mode.

An alternate approach would be to design your application to call function 7 of Extended Services to suspend until a key is pressed or the smallest time interval you are displaying has passed. When this call returns due to a time out, update the clock display and re-issue the suspend call. Displaying time to a 1 minute resolution would certainly be preferable to using a 1 second resolution.

Direct Control of DIS Mode

Designing your programs to hang for a key in as many situations as possible will prevent having to deal with the extra interfacing details covered in this section. However, there are times when this approach is not possible. When you are coding in a high level language you don't always have explicit knowledge of or control over how your target code will interface with the operating system.

Your compiler and its associated development package may generate calls to library functions which use a polling form of console input instead of hanging for a key. This could be done to support pseudo- multitasking features from within a program such as background printing, mouse polling, or the maintenance of an on-screen clock.

There may also be times when you have total control over the method of calling for console input but need to use a polling method because there is no other way to satisfy the requirements of your design specification.

Function 1CH of Extended Services interface gives programs the ability to control MOS's automatic disable feature. If a particular program requires that this feature be inactive for proper operation, it can save the original state of this feature and then insure that it is turned off for the duration of the program's execution.

Saving the original value is required so that the original state can be restored just before your program terminates. In addition, manipulating a system variable in this way does extract a price. In order to be robust, a program which changes a system flag or feature should also intercept the vectors for INT 23H and INT 24H. This is because a program may be able to be terminated through a Control-C (INT 23H) or by choosing Abort from a critical error prompt (INT 24H).

As an alternative approach, consider the following case. Suppose you are designing the world's greatest word processor and you want to include a print-while-you-work feature. To accomplish this, you setup a loop which checks for an available keystroke and then for any waiting printer data. Whenever printer data is ready, this loop sends a character to the printer, checks for a keystroke, sends to the printer, etc.

If DIS mode were not used with such a program, whether activated manually by the user or automatically by the application, this program would be a CPU "hog" when it was idle. One way to avoid this situation would be to design the program to call a function which hangs for a key when no print job is waiting and use the polling loop method when printer data is available.

Working with INT 8H and INT 1CH

When an applications program, TSR, or device driver establishes an intercept of INT 8H or INT 1CH, that intercept will not get called on each INT 8H which occurs when multiple tasks are active. Any program code which is loaded into task memory space will not get all of the actual timer ticks. Some share of timer imterrupts will occur while the task involved is temporarily switched out during the time slice of another task.

There is also a condition where no timer intercepts will be called. When the IRQ0 timer tick interrupt occurs during certain points within MOS's task switching process, while the INT 8H handler within the ROM BIOS will be called, any intercepts will not be.

By allowing IRQ's to occur during a task switch, MOS's interuppt latency is improved. However, a consequence of this is that intercept calls are sometimes skipped to prevent user code from getting control when the system is in a transient state.

Therefore, do not design timer intercept routines which expect to be called on each and every timer interrupt. This means that when using a timer intercept to measure time, counting interrupts could lead to false measurements. One alternative would be to read the timer count value in the BIOS data area (a double word at 40:6C) and calculate what value this field will have when your time period has expired.

When a poll of the BIOS timer count yields a value which meets or exceeds your predicted value, the time interval has expired. In most cases, it is acceptable to use only the least significant word of this field to simplify the calculations.

Testing for a "meets or exceeds" condition is important here. Since task may not be switched in during each timer tick interrupt, testing for an exact match can result in a missed event. The timer interrupt during which the right value occurs in the BIOS timer count may be given to some other task.

One drawback of using the time count value located within the BIOS data area is that it could be changed while your program is in the middle of a time measurement process. Once the timer's value has been recorded at the start of the time interval to be measured, if someone uses the TIME command within another task to change the software clock, your reference has lost its meaning.

Note that INT 8H intercepts will be called at the normal 18 ticks per second rate regardless of whether or not the MOSADM TMFACTOR command has been used to refine the granularity of task switching.

Using MOS's Services for Time Measurement

In the case where you need to program a time delay which does nothing other than wait for a certain interval, you should use function 7 of MOS's Extended Services to give up the CPU for that interval. This will not only provide you with a simple way to measure time (MOS uses a private counter and gets every INT 8H interrupt call) but it will also improve the overall system throughput.

In addition to its time measurement capability, function 7 can also be made to return when other events such as a keystroke or an IRQ are detected. See Chapter 8 for the documentation on this system call's entry and exit parameters, and Chapter 13 for programming examples.

Working with INT 9H

Handling INT 9H intercepts within a multitasking multiuser environment can present certain compatibility problems. The primary reason a program establishes an intercept of INT 9H is to be able to input and process the scan codes generated by the computer's keyboard controller chip. This is typically done by executing an IN AL,60H instruction from within the ISR.

The problem with this practice is that the master console is the only workstation which this is actually valid for. If a program which is running in a background or workstation task establishes an INT 9H intercept, a special type of emulation must be done in order to prevent problems. Otherwise, a TSR loaded within a workstation task might respond when its hot key happens to be used at the master console.

This emulation requires the use of the I/O trapping capability of the 80386 CPU, so the $386.SYS memory management driver must be installed. When MOS is running on system with an 80286 or lesser processor, this form of emulation is not possible and certain compatibility problems may arise.

Beginning with release 4.00, MOS supports intercepts of function 4F of INT 15H. Each time a scan code is about to be processed, the MOS kernel will issue an INT 15H call with AH = 4FH and the scan code in the AL register. If an intercept routine determines that it wants the scan code for its own purpose, it should process the code and then manipulate the flags image on the stack so that when an IRET is done back to the caller, the caller will find the carry flag clear.

When your intercept routine finds that AH is not 4F or that the scan code is not the one it wants, it should pass the call on to the ISR to which the INT 15H vector originally pointed. This must be done just in case another intercept routine is installed which might want to acquire the scan code.

Another interesting use of this method is in scan code translation. Having your intercept handler change the value in AL and return with the carry flag still set (as it is when the call is made), will cause the kernel to use your new value in place of the original.

Making use of this INT 15H service allows for orderly management of scan code intercepts without the compatibility problems an INT 9H intercept can pose when I/O trapping is not available. This feature is available beggining with release 4.00 of MOS. NOTE that it IS AVAILABLE under MOS regardless of the of configuration information returned by the ROM BIOS when an INT 15H function C0H call is made.

Intertask Communication

MOS has two ways of providing intertask communication. The $NET-BIOS.SYS driver and the $PIPE.SYS driver. $NETBIOS provides a series of calls which transfer messages from one task to another. $PIPE.SYS is treated as a file and tasks open the filename specified on the device=$PIPE.SYS statement in the CONFIG.SYS file. The application may then read and write to the file via the handle function calls.

A pipe driver can be used to pass data between tasks or it may be used as a syncronization flag since a task can be made to suspend until data is available from the pipe. To simplify programming, you may want to install one pipe for each communications direction.

Note that since the device name used by the pipe driver may be selected, you must be sure not to assign a name which is already in use as the name of a file or directory. Doing so will render that file or directory inaccessible.

Other methods you may wish to consider are actually using a file as a common message storage facility, or writing your own device driver to handle message read/write arbitration. The device driver method is preferred because drivers are in the SMP and accessible to all tasks, and they are faster than writing to disk files. Remember to use the wait calls if necessary.

See Chapter 12 for more information on $NETBIOS.SYS and the $PIPE.SYS drivers.

TSR Programs

The following points should be kept in mind when writing terminate-and-stay-resident programs:

1. Use INT 21H function 31H to go resident.

2. Use the get/set interrupt vector calls, 35H and 25H for saving/setting interrupt vectors. Be sure to pass control on to the old ISR.

3. Use INT 10H calls or write to the CON device for video output. Do NOT use direct video RAM writes. Direct video writes are faster in a single-user environment. However, in a multi-user system, overhead for processing direct video-RAM writes is greater than the first two methods, especially as the number of users on the system increases. The program will also be incompatible in non-memory managed systems.

4. Do not continually reset your INT vectors. Take them once. MOS will retake the vector(s), and pass control on to your interrupt service routine(s) when appropriate.

Also, refer to the sections on working with INT 8H and INT 9H earlier in this chapter.

General Product Design Considerations

When MOS boots in a system with paging-capable memory management support, the kernel will attempt to relocate its system overhead components into the upper memory area between C0000 and F0000. The effect of this relocation is that more of the base 640K memory area is left free for use by applications.

In order for this relocation to be as effective as possible, whatever ranges of the C0000 to F0000 memory space are available should be in large contiguous spans.

When a hardware product needs to use upper memory, it is, of course, best to minimize the size of this memory usage as much as possible. In addition, making the location of the memory range adjustable will allow the user to configure their system for the greatest remainder of contiguous free memory space.

Avoid programming techniques which scan through the PC-DOS kernel or its command processor looking for internal variables or sequences of code. Such non-portable practices will only limit the market for your product.

Applications which must use a serial port through direct access should be configurable with respect to the serial port address and IRQ. Making use of MOS's extended INT 14H interface will insure compatibility and maximize overall system throughput as opposed to placing IRQ handlers in switched memory.

Regarding the placement of IRQ handlers, put them within a device driver whenever possible. When IRQ handlers are located within switched memory (task memory), a pair of task switches will sometimes be required in order to process the event.

If the IRQ happens to occur when the target task is the currently active one, then MOS's IRQ management logic can call the ISR. However, if another task happens to be switched in, the target task must be switched in, the ISR called, and then another task switch done to restore the original machine state.

In a high frequency IRQ situation, this can cause a thrashing effect. For the sake of overall system efficiency, it is always better to place IRQ handlers in non-switched memory rather than within the task. An SMP loaded device driver is non-switched.

An important note when designing an ISR within a device driver is that the state of the task RAM and its associated context area cannot be guaranteed. Device driver ISR's must not make any references to code or data located within switched memory (task memory).

Task Unregistration

Resident code modules such as device drivers and TSR programs may, in some cases, require notification that a task is being removed. An example of this would be a network communications device driver which allocates I/O buffers from a pool for each task making use of its services. If resources allocated to a task are not freed when the task is removed the pool will soon become exhausted.

The TCBUNREG field exists within each task's TCB to provide a hook into MOS's task removal process. Device drivers and TSR programs that need to deallocate resources when a task is removed can hook this vector to point to an unregister handler within the driver.

This doubleword field is located at offset 07A6H within the TCB data structure. It is used as a far pointer and is initially set to 0 to indicate that no unregistration routines exist for the task. When a task is being removed, if the task's TCBUNREG pointer is not zero, a far call is made to the address held by this pointer.

A driver establishes its hook into the unregistration calling chain by first recording the current value of the TCBUNREG far pointer in a local variable (one per task being managed). Then the driver must write the address of its own unregistration handler into this field. Extended Services function 2AH is ideal for this exchange.

When a driver's unregistration handler receives control, the DS register contains the segment/selector of the TCB for the task being removed. The driver should use this to identify any resources allocated to a task so that they may now be freed.

Once all processing is done, the entry registers should be restored. Then, if the previous value of the TCBUNREG field is non-zero, a far jump must be done to that address to pass control on to other unregistration handlers. If the previous value is zero then the end of the chain has been reached and a far return should be executed.

CHAPTER 7: FILE I/O & STANDARD DEVICES

File and Path Names

MOS file names consist of two parts. The first part consists of from 1 to 8 bytes which make up the filename. The second part consists of from 1 to 3 bytes that make up an optional file extension. The combination of the filename and the extension give you the ability to create unique file names.

Normally, the filename provides the uniqueness and the extension provides classification. This classification can be used to give information as to what the file contains. For example, the .BAT extension designates a batch file. The .COM and .EXE both indicate that the files are programs.

Filenames and extensions may be created using any of the alphanumeric ASCII characters. The characters restricted from use in filenames/extensions are: < > ¦ * ? \/ , " . [ ] = : ; +, and any character whose ASCII value is less than 20H.

A path may be thought of as a "fully qualified" filename or directory name. A path name consists of a series of directory entries separated by a backslash (\) that tell where in the directory structure a file may be found. The path name may be up to 64 characters long. This includes the filename and extension. The path must terminate with a NULL character (00H). A path in an assembler program might be as follows:

```
    path db  '\main\sub1\info.dat',0
```

This example specifies the path starting at the root directory and preceding down the main and sub1 sub-directories to the file info.dat. Note the null character terminating the filename string. Paths may be used with all "handle" related file calls.

The leading backslash begins the path at the root directory. With no leading backslash, the path begins in the current directory. The path may also contain a drive identifier and its associated colon. For example, the filename string:

```
path db   'd:\jeff\joe\',0
```

starts the path at the root directory of the logical D drive and proceeds down the jeff and joe sub-directories. Again, note the null terminator. Spaces are treated as significant characters in a path name and may be used. The following filename string is valid:

```
path db   '\jeff\word p\letter.doc',0
```

File Handles

All files are referenced by their associated name. To make file access easier for applications programmers, MOS provides file handles. A handle is a 16-bit unsigned integer which MOS returns to an application program when the program asks MOS to create or open a file or device. Think of a file handle as a shorthand method of referencing a file or device.

Handles are the preferred method of doing file I/O. "FCB's" are supported for compatibility only and the associated function calls will not be documented in the System Calls chapter. Handles provide the necessities of file I/O in a multi-user environment. "Handle" functions are required for effective file and record locking. FCB's do not provide this facility.

Handles 0 through 4 are reserved handles that reference standard system devices. 0 is the standard input device. 1 is the standard output device. 2 is the standard error device. 3 is the standard auxiliary device. And 4 is the standard printer device.

Two things applications developers should bear in mind are: 1) Do NOT depend upon a file handle being within a certain range. It is not unreasonable for a handle to be greater than 0FFH (given a relocated PSP handle table - see below). 2) The allowed number of opened files is restricted only by the SMP (System Memory Pool) size. This is because all information relating to files is kept there (except I/O buffers).

One requirement for having a large number of handles open at once is that the existing PSP handle table be copied to a new, larger location and the PSP handle table pointer be updated to point to the new table. Refer to Chapter 9 for more details on the PSP structure.

File and device I/O has two modes, binary and ASCII. The default mode for files is binary. The default mode for devices is ASCII. The differences, pertaining primarily to the console device, are as follows:

Binary Mode File/Device Reads

The control characters (C,S,P) are not monitored. They go straight to the I/O buffers. Nothing is echoed to the standard output device.

The requested byte count is read and completion of the I/O request is immediate when the count request is met or end of file is reached.

Binary Mode File/Device Writes

The control characters (C,S,P) are not monitored. They go straight to the file or device. Nothing is echoed to the standard output device. The number of bytes specified to be written are written. Tabs are not expanded into spaces.

ASCII Mode File/Device Reads

The control characters (C,S,P,Z) are checked. Ctrl-Z is an end-of-file marker. Characters are echoed to the standard output device and, if the character to output is a tab, it is expanded into spaces. The tab character, however, remains in the input buffer.

The input may or may not contain the carriage return/line feed characters. If the number of characters entered extends beyond the number requested, only the number requested are returned. All others are ignored. If a carriage return is entered before the number requested, whatever is entered in the buffer, including the carriage return, is returned. The input line read may be edited using MOS's command buffer editing features.

ASCII Mode File/Device Writes

The control characters (C,S,P,Z) are checked for during the write. Ctrl-Z is an end-of-file marker and, if encountered during the write, causes the write operation to stop. The number of bytes actually written is returned to the user.

Tabs are expanded into spaces. Control characters are printed as defined in the character set for the PC.

The Console Device

MOS's console driver (CON) is a standard character device driver. It provides an interface between MOS and the computer's monitor and keyboard. The driver communicates with the monitor via the INT 10H calls and the keyboard via the INT 16H calls.

There are a few additional features. The driver is re-entrant. It maintains video state information and keyboard state information for each task on the system. This driver handles all consoles on the system, not just the master console. The CON driver calls the terminal device drivers via INT 10H for screen output. When the CON driver gets control it saves the keyboard and video state information in the TCB (Task Control Block) addressed by the SS register. The proper TCB is switched in by MOS before the CON driver gets called. The scratch area in the TCB for the CON driver is structured as follows:

Offset	Description	Initial Value
013AH	Current attribute	7
013BH	ANSI mode flag	'Y'
013CH	Quotation parsing toggle	0
013DH	ESC pending flag	'N'
013EH	Temporary cursor position	0
0140H	Extended Key Flag	'N'
0141H	Buffer for extended key	0
0142H	Next routine to process ESC seq	0
0144H	ESC sequence buffer pointer	0
0146H	ESC sequence buffer - 60 bytes	0

These fields are set to their initial values by MOS when the task is initialized. The ANSI mode flag is 'Y' or 'N' depending upon whether the driver is interpreting ANSI ESC sequences or not.

The quotation toggle is specific to this driver. It is set to non-zero when a quoted string is entered and it's cleared to zero when the ending quote is found. The ESC pending flag is used to gain speed by not sending an ESC until the entire string is interpreted. The Extended Key Flag is used to indicate an extended key. That character is stored in the buffer for extended keys.

The final three fields are used in processing ANSI ESC sequences. The "next routine" field is the address of the routine to call to process the ESC sequence. The buffer pointer points to the location in the ESC sequence buffer where the next character received will be stored.

These fields within the TCB are used solely by the console driver. This is how the MOS console driver uses them. A replacement console driver may use this area as desired.

ANSI ESC Sequences

All ANSI ESC sequences begin with the same two characters: the ASCII ESC (1BH) followed by a left bracket, [(5BH). The lead-in characters are followed by optional parameters. Multiple parameters must be separated by a semi-colon. The string ends with a terminator character.

The following table lists the supported terminators, what they do, and optional parameters.

Terminator Characters

Char	Description	Options
H	Positions cursor	r;c ESC[r;cH
A	Cursor up	p ESC[pA
B	Cursor down	p ESC[pB
C	Cursor right	p ESC[pC
D	Cursor left	p ESC[pD
f	same as H	r;c ESC[r;cf
s	Save cursor position	ESC[s
u	Restore cursor position	ESC[u
J	Clear the screen	ESC[2J
K	Clear to end of line	ESC[K
m	Set graphics rendition	a ; , ; am ESCpm

For the previous table, r = row, c = column, p = a repeat count. If no options are specified, a default of 1 is assumed, and the cursor is placed in the home position for the H and f terminators.

The ’a’ option for "set graphics rendition" is the graphics attribute to be used, per the following table:

Graphics Attributes

Number	Attribute	
0	White on black - normal intensity	
1	High intensity	
4	Underline	
5	Turn blinking on	
7	Inverse video on	
8	Invisible	
30	Black	Foreground
31	Red	"
32	Green	"
33	Yellow	"
34	Blue	"
35	Magenta	"
36	Cyan	"
37	White	"
40	Black	Background
41	Red	"
42	Green	"
43	Yellow	"
44	Blue	"
45	Magenta	"
46	Cyan	"
47	White	"

Other Standard Devices

The following list represents all the device drivers automatically loaded with MOS:

1. Printer devices - LPT1 through LPT3, PRN
2. Serial devices - COM1 through COM24, AUX
3. The NUL device, the CLOCK$ device and the CON device.
4. The diskette/fixed disk device drivers.

This page intentionally left blank.