Microsoft Multitasking MS-DOS Product Specification

SYSTEM CALLS

October 30, 1984

ABSTRACT

This document describes the MS-DOS 4.0 system calls that have been changed or added since the MS-DOS 2.11 release.

1. INTRODUCTION

The purpose of this document is to introduce the new MS-DOS 4.0 system calls, and to describe the changes made to existing MS-DOS calls.

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. CONVENTIONS

In this document, calling sequences for the system calls show the function numbers for the call symbolically with the function number for the system call in a comment. These symbolic names for the system call function numbers are given in the include file SYSCALLS.INC. It is recommended that the symbolic names be used for readability. Other include files which define symbolic names and data structures for specific system calls will be mentioned under the specific calls to which they apply.

Some new system calls are invoked via the traditional **INT 21** interface, others are provided only in library form. The library routines are compatible with assembly language, Microsoft "CMERGE" C, Pascal and Fortran. Other languages may use these functions by creating special invocation subroutines just like the ones they currently must use for **INT 21**-based functions. The calling sequence for the library routines is described herein. The library routines fully conform to the MS-DOS 4.0 memory management requirements and may be used by conforming and non-conforming applications.

3. DEFINITIONS

Process

A process (or task) is a program and all the variables associated with it. This includes a set of per-process data maintained by the system.

Process ID (PID)

The Process ID used to identify a process is a 16-bit number unique to every process in the system.

Command Subgroup ID (CSID)

The Command Subgroup ID is a number that is derived from the PID. It is used to identify all processes that have been spawned by another process, either directly or indirectly.

Zombie Process

After a process has terminated, it is placed into a special state called *zombie* to preserve its exit code. This exit code may be examined later by the process that spawned it.

4. PROCESS CONTROL CALLS

Below is a list of process control calls. These calls are described in detail in the following sections.

Exec	Start a new task.
Wait	Return child termination code.
CWait	Wait for child termination.
Freeze	Stop a process.
Resume	Restart a process.
Sleep	Delay process execution.
Kill	Terminate a task.

4.1. EXEC—Start a New Task

Exec is used to run other processes. These are called "child" processes because they were created by the current process, inherit some of the environment (such as open file handles) from the "parent", and are susceptible to some forms of parental process control.

Calling Sequence:

```
        MOV     AH,Exec                 ; 4Bh
        MOV     AL,Subfunction          ; (AX) = function code
        LDS     DX,Name                 ; (DS:DX) = ASCIIZ pathname
        LES     BX,Parmblock            ; (ES:BX) = Addr of Parameter Block
        MOV     CX,Mode                 ; (CX) = mode flags
        INT     21H                     ; Call MS-DOS
```

Returns:

```
        CARRY SET IF ERROR
        ELSE
             AX = PID/CSID of child
```

Exec supports several subfunctions, described below.

4.1.1. Subfunction 0—Start Synchronous Task

This mode of the **Exec** call is compatible with the MS-DOS 2.0 **Exec** call. It will start a new task, but the parent is suspended until the child terminates. Refer to the 2.0 *MS-DOS Programmer's Reference Manual*, Function 4BH, for details. Note that if a task uses this call to start and wait for a child, then it should use the **Wait** call to retrieve the termination code for the child. Using the **CWait** call will return an error as there is no child to wait for. The *Mode* argument is not used.

4.1.2. Subfunction 1 – MS-DOS Reserved.

This function is reserved for use by Debug.

4.1.3. Subfunction 2 – MS-DOS Reserved.

4.1.4. Subfunction 3 – Load Overlay

Same functionality as in MS-DOS 2.0. Refer to the MS-DOS 2.0 system calls documentation for details. The *Mode* argument is not used.

4.1.5. Subfunction 4 – Start Asynchronous Task (Exec & Go)

Same format and operation as Function 0, except that the parent task continues to run in parallel to the child task. If *Mode* = 0, when the child terminates it is placed in a zombie state to preserve its return code until the parent executes a **CWait** system call.

If *Mode* = 1, then the child is not placed in a zombie state when it terminates; it disappears and its result code is discarded. If a process executes a child using this mode, there is no way for the parent to wait for the child.

4.2. WAIT—Return Synchronous Child Termination Code

This is the same call as in MS-DOS 2.0.

```
Calling Sequence:

        MOV     AH,Wait             ; 4Dh
        INT     21H

Returns:
        AX = Return code from Child
```

Note that this call should only be used with **Exec** subfunction 0. This call will return garbage if used with any other form of **Exec**.

4.3. CWAIT—Wait for Child Termination

CWait is a new system call that waits until a child task has terminated, then returns its PID and termination code. The only way to verify that the return code is from a specific child is by checking its PID. If no children were started, then **CWait** returns with an error. If no children have terminated, then **CWait** will wait until one terminates before returning to the parent.

CWait is normally used to wait for an entire group of processes belonging to a command subtree. A program that invokes a child task is not aware of the details of the child's implementation. The child task may well create a grandchild task to handle some of the work. It does the parent little good to know that his direct child has completed and exited; the parent needs to know that the entire family of processes, the command subtree, has completed.

Children started as orphans (**Exec** call function 4 mode 1) are unknown to **CWait**, and do not otherwise affect its operation.

```
Calling Sequence:

        MOV     AH,CWait            ; 8Ah
        MOV     BL,Range            ; =0 if to wait for command subtree
                                    ; =1 if to wait for any child
        MOV     BH,Flag             ; =0 to suspend if children exist
                                    ;    but are not dead yet
                                    ; =1 to return if no child currently
                                    ;    dead ('C' clear, (BX) = 0 )
        MOV     CX,PID              ; PID of head of command subtree
        INT     21H

Returns:
        CARRY SET IF ERROR
                AX = No_child
        ELSE
                AX = Return code from Child
                BX = PID of child
```

4.4. FREEZE—Stop a Process

This call stops a task or a command subtree by removing it from the run queue until a **Resume** call is made for that PID. The task may not be stopped immediately because it may have some resources locked that should be freed first.

Calling Sequence:

```
        MOV     AH,Freeze               ; 81h
        MOV     BX,Flag                 ; =0 if to freeze for command subtree
                                        ; =1 if to resume specific task
        MOV     CX,PID                  ; PID of head of command subtree
        INT     21H
```

Returns:

```
        CARRY SET IF ERROR
                AX = No_such_process
```

4.5. RESUME—Restart a Process

This call restarts a process previously frozen via the **Freeze** system call.

Calling Sequence:

```
        MOV     AH,Thaw                 ; 82h
        MOV     BX,Flag                 ; =0 if to resume for command subtree
                                        ; =1 if to resume specific task
        MOV     CX,PID                  ; PID of head of command subtree
        INT     21H
```

Returns:

```
        CARRY SET IF ERROR
                AX = No_such_process
                   = Not_frozen
```

4.6. SLEEP—Delay Process Execution

The **Sleep** call suspends the current task for the specified time period. The actual time it is asleep may be off by a clock tick or two, depending on the execution status of other tasks running in the system. If the time is 0, then the process will forego the remainder of its CPU time-slice, but will be scheduled normally for its next slice. Otherwise, the time is given in milliseconds and will be rounded to the resolution of the scheduler clock.

Calling Sequence:

```
        MOV     AH,Sleep                ; 89h
        MOV     CX,Time
        INT     21H
```

4.7. KILL—Terminate a Task

This call terminates a task or command subtree and can be used to terminate the task that issued the call. Assuming that the task is not an orphan, the task's parent will get a "process killed" termination code returned when it does a **Cwait** call.

```
Calling Sequence:

        MOV     AH,Send_Signal          ; 8Dh
        MOV     AL,SIGTERM
        MOV     BH,0
        MOV     BL,Action               ; =0 to kill command subtree
                                        ; =1 to kill just specified process
        MOV     DX,PID
        INT     21H

Returns:
        CARRY SET IF ERROR
                AL = error_invalid_function
                        Invalid Action.
                AL = error_invalid_handle
                        No processes matched the PID given in DX
                        or some process had error action specified
                        for the signal.
```

5. PROCESS INFORMATION

The process information calls are:

GetPID	Return Process ID.
Priority	Get or set task priority.

These are discussed in more detail in the next sections.

5.1. GETPID—Return Process ID

This call will return a process's PID and the PID of the task that spawned it. The PID may be used to generate uniquely named temporary files (see also **CreateTemp**), or for communication via signals.

Calling Sequence:

```
    MOV     AH,GetPID                   ; 87h
    INT     21H
```

Returns:

```
    AX = PID
    BX = Parent process's PID
    CX = CSID
```

5.2. PRIORITY—Get/Set Task Priority

This call allows the caller to learn or change the priority of a process. A process's priority can range from –15 to +15, with 0 being the normal priority. The argument to this call specifies a signed delta-value; that value is added to the current priority, the result is restricted to the legal range, and the new value is returned. A delta-value of 0 thus returns the current priority.

Priority –15 means that this task will run only when there is no other runnable task; priority +15 means that this task will receive whatever CPU resources it requires. (In order to prevent system starvation, any runnable task receives at least 1/10th of a second service at least every 3 seconds.) It may not be possible to raise the priority of a process.

Calling Sequence:

```
    MOV     AH,Setpri                   ; 8Eh
    MOV     AL,Priority_delta
    MOV     CX,PID
    INT     21H
```

Returns:

```
    CARRY SET IF ERROR
            AX = No_such_process
               = Invalid_priority
    ELSE
            AL = Process priority
```

6. INTRAPROCESS CONCURRENCY CALLS

MS-DOS 4.0 provides calls to allow the creation, control, and termination of multiple *threads* of execution within a single task. A high-efficiency memory-based semaphore mechanism is provided to control access by multiple threads to critical sections and monitors within the task.

The **Thread** system call allows a program to create additional threads of control and thus be "simultaneously" executing its code in several spots at once.

A thread is *not* another process. If a process is running with two threads they are nearly identical: if one thread issues an **Open** system call and gets back handle #4, the other thread can do a **Read** on handle #4 to read that data. The only thread-specific information maintained is the register contents. Any thread may make system calls and "speak" on behalf of the process. If any thread issues the **Exit** system call then the process will terminate, therefore terminating all its threads.

A task's initial thread is called *thread 0*. This thread cannot be terminated via the **Thread** system call, and is the thread that is interrupted in response to a system *SIGNAL*.

The intraprocess concurrency calls are:

Thread	Create or Terminate a Thread.
CritEnter	Block on memory-based semaphore.
CritLeave	Release memory-based semaphore.

These calls are detailed in the following sections.

6.1. THREAD—Create or Destroy Process Thread

Calling Sequence:

```
        MOV     AH,148          ; THREAD (94h)
        MOV     AL, <flag>      ; (0 to create, 1 to terminate)
        MOV     BX, <offset of new stack in DS segment>
        MOV     CX,0
        INT     21H
```

Returns:

```
  If (AL) = 0  (create new thread)

        CARRY set if error
                AX = Too Many Threads
        CARRY clear if OK
                (AL) = 0 if parent thread
                (AL) = 1 if child thread (using new stack)

  If (AL) = 1  (terminate thread)

        CARRY set if error
                AX = Illegal Termination (attempt to terminate thread 0)
        Does not return if no error
```

6.2. CRITENTER and CRITLEAVE—Block Process on RAM-Semaphore

CritEnter and **CritLeave** are two new system libary routines that are used to interlock access to critical data structures that are shared between two or more threads or processes. They implement a classic semaphore mechanism and can therefore be used to synchronize activities as well as protect shared data areas.

CritEnter and **CritLeave** operate by means of a flag word in RAM. The flag word should be initialized to zero if the semaphore is to be initially unset (unlocked), and initialized to 1 if the semaphore is to be initially set (locked). A near pointer to the semaphore word is passed to both **CritEnter** and **CritLeave**. (Alternative versions called **FCritEnter** and **FCritLeave** take far pointers to the semaphore word, but are otherwise identical.)

When **CritEnter** is called, it checks the status of the flag word. If it is unset, then **CritEnter** sets it and returns immediately to the caller. If the flag is set, **CritEnter** blocks the thread until the flag becomes unset, then tries again. When the thread is done with the protected resource, it calls **CritLeave**. **CritLeave** then unsets the flag word and starts any threads that were blocked waiting for that particular flag word.

CritEnter and **CritLeave** are similar to **Waitsem** and **Sigsem** (see Section 7, "Interprocess Communication"), except that **CritEnter** works via a RAM location whereas **Waitsem** works via a file system entry. **CritEnter** is very efficient for closely coupled threads and processes that share memory; **Waitsem** is useful for interlocking uncoupled processes.

```
Calling Sequence:

 in C

        static word flagword = 0;

    CritEnter (flagword);
        .
        .
    CritLeave (flagword);

 in Assembly

        flagword .word  0                     ; in a data segment
        .
        .
    MOV     AX,OFFSET DS:flagword
    PUSH    AX
    CALL    CritEnter
        .
        .
    MOV     AX,OFFSET DS:flagword
    PUSH    AX
    CALL    CritLeave
```

7. INTERPROCESS COMMUNICATION CALLS

Pipes reside in a virtual directory called \PIPE. All named pipes must begin with the string \PIPE\. See the document *Microsoft Multitasking MS-DOS Product Specification OVERVIEW* for a discussion of pipes.

Pipes are different from files because writes to a pipe will not be interspersed. A process issuing a write to a pipe will be blocked until it can write all of the data specified to the pipe. No other process may write to that pipe until the write in progress completes. Any process that attempts to write to the pipe is blocked.

Named pipes are created and accessed via the **Create** and **Open** system calls. They are used to communicate with other non-related processes. *Anonymous* pipes are created via the **Pipe** system call. They are used to communicate between parent-child or sibling-related processes. All pipe I/O is done via the standard **Read** and **Write** system calls. All pipes are closed via the standard **Close** system call.

The interprocess communication calls are:

Pipe	Create an anonymous pipe.
CreatMem	Create a shared memory area.
GetMem	Access a shared memory area.
ReleaseMem	Release access to a shared memory area.
Creatsem	Create system semaphore.
Opensem	Open system semaphore.
Waitsem	Check a system semaphore.
Sigsem	Release a system semaphore.

These calls are detailed in the following sections.

7.1. PIPE—Create a New Pipe

Pipe will create a new anonymous pipe. It returns two handles, one for reading the pipe, the other for writing. The size parameter is advisory only. If the size parameter is zero, then the pipe will be created with the default size.

Calling Sequence:

```
        MOV     AH,MakePipe
        MOV     CX,Size
        INT     21H
```

Returns:

```
        CARRY SET IF ERROR
              AX = Insufficient memory
        ELSE
              AX = read handle
              BX = write handle
```

7.2. CREATMEM—Create a Shared Memory Area

This call is used to obtain a memory region for private use between two or more tasks. If a shared memory region with the same name exists already, the creation fails.

Shared memory areas have no underlying structure. It is up to the cooperating processes to define their own formats and protocols.

```
Calling Sequence:

        MOV     AH,CreatMem             ; 84h
        MOV     CX,Size
        LDS     DX,Name
        INT     21H

Returns:
        CARRY SET IF ERROR
            AX = Invalid_name
            AX = Not_enough_memory
        ELSE
            BX = pointer to shared memory (paragraph #)
```

7.3. GETMEM—Obtain Access to a Shared Memory Area

This call is used to obtain access to an existing shared memory area. Before executing this call, the shared memory area must have been created through the CreatMem system call.

```
Calling Sequence:

        MOV     AH,GetMem               ; 85h
        LDS     DX,Name
        INT     21H

Returns:
        CARRY SET IF ERROR
            AX = Invalid_name
        ELSE
            BX = pointer to Shared memory (paragraph #)
            CX = Size in paragraphs of the Shared Memory
```

7.4. RELEASEMEM—Release Access to Shared Memory

This system call terminates the access of a process to a shared memory area. If the reference count for the memory area is reduced to zero, the area is released and the memory region deallocated.

```
Calling Sequence:

        MOV     AH,ReleaseMem           ; 86h
        LDS     DX,Name
        INT     21H

Returns:
        CARRY SET IF ERROR
            AX = No_such_name
```

7.5. CREATSEM—Create System Semaphore

Creatsem creates a binary semaphore pseudo-file for subsequent use by **Opensem**, **Waitsem**, and **Sigsem**. The semaphore can then be used to manage mutually exclusive access to a resource, shared memory, or a critical section of a program. The semaphore is a "pseudo-file" in that its name takes the form of a file in subdirectory "\SEM\", although such a subdirectory does not exist. MS-DOS 4.0 keeps the semaphore names in memory. A future release of MS-DOS will support an extended file system that will allow the semaphores to be stored in a disk subdirectory. This will allow the user to control access to semaphores in the same way that he will be able to control access to individual files.

A different semaphore facility, **CritEnter** and **CritLeave**, was described previously. They implement a high-speed semaphore mechanism but require that the using processes have shared access to a flag word. This is called the RAM-semaphore facility.

Creatsem, **Opensem**, **Waitsem** and **Sigsem** are called the "system semaphore" facility. They use a (pseudo) file system entry instead of a RAM location. This gives them much more flexibility then the RAM-semaphore mechanism because:

1) They can be used by processes which share no memory.
2) They can be used via a network by processes which are running on another machine.
3) Access to system semaphores is protected via the same mechanism which protects access to files.

Creatsem is used to initially create a system semaphore and set its value to *clear*.

Calling Sequence:

```
        MOV     AH,syssem
        MOV     AL,0                    ; Creatsem
        MOV     BX,0777H                ; (BX) = 0777
        LDS     DX,semname              ; (DS:DX) = address of name string
        INT     21H
```

Returns:

```
        CARRY SET IF ERROR
            (AX) = error code
                   illegal name format
                   semaphore already exists
        CARRY clear if OK
            (AX) = semnum
```

Semname is the name of the semaphore. It must be in the format of an MS-DOS file in a subdirectory called \SEM\, such as \SEM\PRINT.LCK.

Semnum is the value supplied to **Waitsem** and **Sigsem** to control the semaphore.

7.6. OPENSEM—Open an Existing System Semaphore

Opensem opens a system semaphore of the specified name and returns the unique semaphore identification number *semnum* used by **Waitsem** and **Sigsem**. **Creatsem** must have been previously called to create the semaphore before it can be opened. Typically the first user of the semaphore creates it via **Creatsem**; subsequent users open it via **Opensem**. **Opensem** does not test or change the value of the semaphore.

Note that, under MS-DOS 4.0, system semaphores reside in a memory buffer rather than on a disk file system. This means that when the last process which has a semaphore open (via **Creatsem** or **Opensem**) exits, the semaphore disappears and must be re-created by its next user.

Calling Sequence:

```
        MOV     AH,syssem
        MOV     AL,1                            ; opensem
        LDS     DX,OFFSET DS:semname            ; (DS:DX) = name string
        INT     21H
```

Returns:

```
        CARRY SET IF ERROR
            (AX) = error code
                    illegal name format
                    semaphore does not exist
        CARRY clear if OK
            (AX) = semnum
```

Semname is the name of the semaphore. It must be in the format of an MS-DOS file in a subdirectory called \SEM\, such as \SEM\PRINT.LCK.

Semnum is the value supplied to **Waitsem** and **Sigsem** to control the semaphore.

7.7. WAITSEM—Block on a System Semaphore

Waitsem checks the status of the specified system semaphore. If the semaphore is clear, **Waitsem** sets it and returns control to the caller. If the semaphore is already set, **Waitsem** blocks the calling thread until the semaphore is released. **Waitsem** then sets it and returns control to the caller.

A flag value can be passed to the **Waitsem** system call to cause it to immediately return an error code to the caller if the semaphore is busy.

Calling Sequence:

```
        MOV     AH,syssem
        MOV     AL,3                            ; waitsem
        MOV     BX,flag                          0 if to block, 1 if not to block
    MOV     CX,semnum                          (CX) = semaphore number
        INT     21H
```

Returns:

```
        CARRY SET IF ERROR
            (AX) = error code
                    illegal semaphore number
                    semaphore is set (only if (BX) = 1)
        CARRY clear if OK
```

Semnum is the value returned by **Creatsem** or **Opensem**.

7.8. SIGSEM—Release a System Semaphore

Sigsem is called to clear ("release") a system semaphore which was previously set via **Waitsem**.

Calling Sequence:

```
        MOV     AH,syssem
        MOV     AL,4                    ; sigsem
        MOV     BX,0
        MOV     CX,semnum               ; (CX) = semaphore number
        INT     21H
```

Returns:

```
        CARRY SET IF ERROR
           (AX) = error code
                  illegal semaphore number
                  semaphore was not set
        CARRY clear if OK
```

Semnum is the value returned by **Createsem** or **Opensem**.

8. INTERRUPT VECTOR CALLS

In MS-DOS 4.0 the 8086/88 interrupt vector table must be managed as an MS-DOS resource. Whenever a program wants to use software interrupts, it must use these calls (an extension of the MS-DOS 2.0 calls) to place the vectors in the table. Interrupt vector usage is identified as being one of two classes by the DOS. Local interrupt vectors are suitable for use with software interrupts within a single process. Global interrupt vectors are suitable for use when dealing with hardware interrupts or when using software interrupts as a means of sharing code or interprocess communication (IPC).

Global interrupts are provided mostly for backward compatibility. The newer mechanisms provided by MS-DOS 4.0 should be used for code sharing and IPC.

The interrupt vector calls are:

GetVector	Get per-process interrupt vector.
SetVector	Set per-process interrupt vector.
Set_Global_Vector	Set system-global interrupt vector.
Unset_Vector	Unset interrupt vector.

These calls are detailed in the following sections.

8.1. GETVECTOR—Get Per-Process Interrupt Vector

Same as in MS-DOS 2.0. See the MS-DOS 2.0 *MS-DOS Programmer's Reference Manual*, Function 35H.

8.2. SETVECTOR—Set Per-Process Interrupt Vector

Same as in MS-DOS 2.0, except that the call may fail if another process has set up the vector as a global vector. In this case, the process issuing the call will be terminated unless it has indicated via **GetExtendedError** that it can tolerate new error conditions. See the MS-DOS 2.0 *MS-DOS Programmer's Reference Manual*, Function 25H.

8.3. SET_GLOBAL_VECTOR—Set System-Global Interrupt Vector

Set_Global_Vector may be used to set an interrupt vector that will be shared by all processes in the system. The call will fail if the vector has already been set as either a local or global vector.

```
Calling Sequence:

        MOV     AH,Set_Global_Vector    ; 8Fh
        MOV     AL,Intr
        LDS     DX,Vector
        INT     21H

Returns:
        CARRY SET IF ERROR
            AX = Fixed_vector
                 Vector_used
                 Global_limit
```

8.4. UNSET_VECTOR—Unset Interrupt Vector

Unset_Vector may be used to reset an interrupt vector that had previously been set as a global vector or as a local vector. The vector will be available for re-allocation by other processes. If the vector had been set as a global vector, the value of the vector will be restored to the value it had before the **Set_Global_Vector** call.

```
Calling Sequence:

        MOV     AH,Unset_Vector          ; 90h
        MOV     AL,Intr
        INT     21H

Returns:
        CARRY SET IF ERROR
         AX = Not set
```

9. FILE MANAGEMENT CALLS

This section describes calls related to the management of files in a multitasking environment.

The file management calls are:

CreatTemp	Create a unique file.
CreatNewFile	Guarantee a new file.
Lock	Prevent access to a range of a file.
Unlock	Release a locked region.
Open	Gain read/write access to a file.

These calls are detailed in the following sections.

9.1. CREATTEMP—Create a File With a Unique Name

CreatTemp generates a unique name and attempts to create a new file in the specified directory. If a file already exists in that directory, then another unique name is generated and the process is repeated. **CreatTemp** is guaranteed to produce a unique name and to avoid any race conditions.

```
Calling Sequence:

        MOV     AH,CreatTempFile         ; 5Ah
        LDS     DX,Directory
        MOV     CX,Attribute
        INT     21H

Returns:

        CARRY set if error
                AX = error code
        CARRY clear if OK
                AX = file handle
                The file name is appended to the directory string
```

Programs that need temporary files must use this system call to generate temporary files to prevent name conflicts in a multitasking environment. The *Directory* string must have at least 13 unused bytes at the end.

9.2. CREATNEWFILE—Guarantee New File

This function is identical to the MS-DOS 2.0 **Creat** system call except that it will fail if the file already exists. A multitasking system must be able to use files and their existence as semaphores. The **CreatNewFile** system call may be used as a test-and-set semaphore.

```
Calling Sequence:

          MOV      AH,CreatNewFile       ; 5Bh
          LDS      DX,Name
          MOV      CX,attributes
          INT      21H

Returns:
      CARRY set if error
               AX = error_file_already_exists
                        The specified file already exists
      CARRY clear if OK
               AX = file handle
```

9.3. LOCK—Prevent Access to a Range of a File

Lock provides a simple mechanism for excluding other processes access to regions of a file. **Lock** will inhibit *all* reads and writes to a file. Tasks that try to access a locked region are suspended until the region is released.

The locked regions may be anywhere in the logical file. Locking beyond end-of-file is not an error. It is expected that the time in which regions are locked will be short; indeed it should be considered an error if they are locked for more than 10 seconds.

Duping the handle will duplicate access to the locks. Access to the locks is not duplicated across the **Exec** system call.

Programs must *not* rely on whether they have read or write access, but should attempt to lock the region desired and examine the resultant error code. Future versions of the **Lock** system call may only advise the program about locking conflicts.

```
Calling Sequence:

          MOV      AX,LockOper SHL 8     ; 5C00h
          MOV      BX,Handle
          MOV      CX,OffsetHigh
          MOV      DX,OffsetLow
          MOV      SI,LengthHigh
          MOV      DI,LengthLow
          INT      21H

Returns:
      CARRY set if error
               AX = error_invalid_handle
                        The handle in BX was not a valid opened
                        handle.
                  = error_lock_violation
                        The  region  (or a piece of the region)
                        specified was previously locked either  by
                        the current process or by another process.
      CARRY clear if no error
               range locked
```

9.4. UNLOCK—Release Locked Region

Unlock releases the lock issued in a previous Lock system call. The region specified must be exactly the same as the region specified in the previous lock.

Calling Sequence:

```
        MOV     AX,(LockOper SHL 8) + 1      ; 5C01h
        MOV     BX,Handle
        MOV     CX,OffsetHigh
        MOV     DX,OffsetLow
        MOV     SI,LengthHigh
        MOV     DI,LengthLow
        INT     21H
```

Returns:

```
    CARRY set if error
            AX = error_invalid_handle
                    The handle passed in BX was not open.
               = error_lock_violation
                    The region  specified  was  not  identical  to
                    one that was locked by that process.
    CARRY clear if OK
            region unlocked
```

9.5. OPEN—Gain Read/Write Access to a File

Open is the same system call as in MS-DOS 2.0, except that the high nibble of the *Access* code now has meaning.

Calling Sequence:

```
        MOV     AH,Open              ; 3Dh
        MOV     AL,Access
        LDS     DX,Pathname
    INT     21H
```

Returns:

```
    CARRY set if Error
            AX = error_invalid_access
                    The low 4 bits of access specified in AL
                     was not in the range 0:2.
               = error_file_not_found
                    The path specified was invalid or not found.
               = error_access_denied
                    The user attempted to open a directory or
                    volume-id, or open a read-only file  for
                    writing.
               = error_too_many_open_files
                    There were no free handles available in the
                    current process or the internal system tables
                    were full.
    CARRY clear if OK
            AX = file handle
```

The following values are allowed for the low nibble of *Access*:

Access Byte—Low Nibble	
Access	**Function**
0	The file is opened for reading. This open will succeed only if the file is not previously open in deny read mode, or in deny read/write mode (see below).
1	The file is opened for writing. This open will succeed only if the file is not previously open in deny write mode, or in deny read/write mode (see below).
2	The file is opened for both reading and writing. This open will succeed only if the file is not previously open in deny read mode, or deny write mode or deny read/write mode (see below).

The high order bit of *Access* indicates whether or not this file is to be inherited by child processes (created through the **Exec** system call). 0 indicates that the child process will inherit the open file, while a 1 indicates that the handle is private to the current process.

The remaining 3 bits indicate the sharing mode for the file:

Access Byte—High Nibble	
Access	**Function**
0	**Compatibility** mode. Any process on a particular machine may open the file any number of times with this mode, provided that the file was not opened previously with one of the four modes below.
1	**Deny read/write** mode. This open sharing mode will succeed *only* if the file is not already open in compatibility mode, or in read or write access by any other process (including the current process). In other operating systems this is known as exclusive mode.
2	**Deny write** mode. This open sharing mode will succeed *only* if the file is not already open in compatibility mode, or in write access by any other process.
3	**Deny read** mode. This open sharing mode will succeed *only* if the file is not already open in compatibility mode, or in read access by any other process.
4	**Deny none** mode. This open sharing mode will succeed *only* if the file is not already open in compatibility mode.

The read/write pointer is set at the first byte of the file. The returned file handle must be used for subsequent I/O to the file.

10. SIGNAL CALLS

Signals provide a software interrupt mechanism that may be used to handle exceptional conditions or simple interprocess communications. The user may install a signal handler that will be called when an event occurs. The signal handler will be called either after a hardware interrupt has interrupted the user process or at the completion of a system call. A few system calls (**Cwait**, **Sleep** and **Read** from a character device) can be interrupted by a signal. The signal handler has the option of returning an error indication from the system call or restarting the system call.

Symbolic definitions of signal numbers and actions are given in the include file SIGDEF.INC. The contents of the file are:

```
NSIG      EQU     16

;         Signal numbers

SIGKB0    EQU     1              ; ^C or user defined key
SIGINTR   EQU     1              ; ^C or user defined key
SIGKB1    EQU     2              ; alternate key intercept
SIGKB2    EQU     3              ; alternate key intercept
SIGMUF    EQU     4              ; special key intercept for MUF
SIGDIVZ   EQU     5              ; divide by zero trap
SIGOVFL   EQU     6              ; INTO instruction
SIGHDERR  EQU     7              ; INT 24 type things
SIGTERM   EQU     8              ; program termination
SIGPIPE   EQU     9              ; broken pipe
SIGUSER1  EQU     13             ; reserved for user definition
SIGUSER2  EQU     14             ; reserved for user definition

;         Signal actions
SIG_DFL   EQU     0              ; terminate process on receipt
SIG_IGN   EQU     1              ; ignore
SIG_GET   EQU     2              ; signal is accepted
SIG_ERR   EQU     3              ; sender gets error
SIG_ACK   EQU     4              ; acknowledge received signal
```

The signal calls are:

Set_Signal_Handler	Handle signal.
Send_Signal	Issue signal.

10.1. SET_SIGNAL_HANDLER—Handle Signal

This call notifies the DOS of a handler for a signal. It may also be used to ignore a signal or install a default action for a signal.

```
Calling Sequence:

      MOV     AH,Set_Signal_Handler    ; 8Ch
      MOV     AL,SigNumber
      MOV     BL,Action
      LDS     DX,Vector
      INT     21H

Returns:
      CARRY SET IF ERROR
              AL = error_invalid_function
                      Invalid SigNumber or Action
      ELSE
              AL = previous action
              ES:BX = previous vector
```

If *Action* is 2, *Vector* must contain the address of a signal handling routine. *SigNumber* gives the number of the signal handled by the routine. If *Action* is 0, a default action is installed for the signal. Usually, this will cause termination of the process if the signal is sent to the process. If *Action* is 1, the signal will be ignored. If *Action* is 3, it will be considered an error for any process to send the signal to this process.

10.2. SEND_SIGNAL—Issue Signal

Send_Signal is used to send a signal event to an arbitrary process or command subtree. **Send_Signal** is normally used to send **SIGTERM** or **SIGIPC**, but can be used to send any signal.

```
Calling Sequence:

        MOV     AH,Send_Signal          ; 8Dh
        MOV     AL,SigNumber
        MOV     BH,SigArg
        MOV     BL,Action               ; =0 for entire subtree
                                        ; =1 for single process
        MOV     DX,PID
        INT     21H

Returns:
        CARRY SET IF ERROR
                AL = error_invalid_function
                        Invalid SigNumber or Action.
                AL = error_invalid_handle
                        No processes matched the PID given in DX,
                        or some process had error action specified
                        for the signal.
```

10.3. Signal Handling Routine

A signal handling routine will be entered with the signal number in AL, an argument value in AH and registers other than AX, FL, SP, CS, and IP having their values at the time that the signal was taken. The value in AH will be the value that a user passed in BH when making a **Send_Signal** call or a system-defined value for signals generated by the system. The signal handler should distinguish between different signals on the basis of the signal number passed. All intercepted signals will be sent to the most recently installed handler.

The signal handler may take any of several actions to continue normal processing. It may do a long return with the carry flag set to cause termination of the process. It may do a long return with the zero flag set, and carry reset, to cause any interrupted system call to return with an error. It may do a long return with the zero and carry flags reset, or an IRET instruction in order to cause any interrupted system call to be restarted. The signal handler may reset the stack pointer to some previous valid stack frame and jump to some other part of the program. In this case, the signal handler should invoke the **Set_Signal_Handler** system call to reset the signal that was taken. The system call may be entered with *Action*=4 in order to reset the signal without affecting the current disposition of the signal.

11. IOCTL CALLS

The **IOCTL** call is used to perform hardware-specific control functions. MS-DOS 2.0 made no further definition; MS-DOS 4.0 expands the specification to include officially defined forms for:

USART Control.

This includes baud rate, stop bits, parity, etc.

TTY Control.

This includes raw/cooked mode, echo on, and echo off, etc.

Screen Switching Functions

These functions include querying the driver for the amount cf space it needs to hold the screen image, as well as command screen saves and restores.

Further, the **IOCTL** call will allow OEMs to define private **IOCTL** values that will not conflict with future MS-DOS developments.

The form of the **IOCTL** call used to control switching between screen groups is detailed in the document *Microsoft Multitasking MS-DOS Product Specification DEVICE DRIVERS.*

12. AUXILIARY CALLS

The MS-DOS 4.0 auxiliary calls are:

GetExtendedError	Return extended error.
SetMaxMem	Set maximum memory allocation.
TimerService	Obtain clock ties.

12.1. GETEXTENDEDERROR—Return Extended Error

With the addition of new functionality to the MS-DOS operating system, many existing programs would not function correctly if a new error was returned for the old system calls. As a result, when new errors occur they are mapped into a simpler MS-DOS 2.0-level error.

Programs that are written from now on are expected to handle errors by noting the error return on the original system call, and then issue the **GetExtendedError** system call. If the program does not recognize the new error, it should use the mapping and behave accordingly.

```
Calling Sequence:

      MOV       AH,59h        ; GetExtendedError
      INT       21H

Returns:
      AX = extended error code
```

12.2. SETMAXMEM—Set Maximum Memory Allocation

This call is provided in order to stop old MS-DOS 2.0 programs from allocating all of memory, and thus not allowing other programs to execute. It takes as argument the maximum size in paragraphs that child tasks can obtain in their initial allocation. It returns the previous value. A *Size* of 0 will not change the current limit, just report its value. The new limit affects the current process' descendents.

For example, if the screen manager wants to run a program that is known to allocate all memory for its use, but does not really need it, it would issue a **SetMaxMem** system call before executing the program.

```
Calling Sequence:

        MOV     AX,(ChildCtl SHL 8) + 1          ; 8301h
        MOV     DX,Size                          ; = 0 if not to change MaxMem
        INT     21H

Returns:
        AX = previous MaxMem
```

12.3. TIMERSERVICE—Obtain Clock Tics

This call links the address passed by the user to the scheduler time code.

```
Calling Sequence:

        MOV     AH,TimServ                  ; 80h
        MOV     AL,Mode
        LDS     DX,Vector
        INT     21H

Returns:
        'C'  clear if no error
             CX = tick interval, in milliseconds
        'C'  set if error
             If Mode=1 then AX = error-code
```

Whenever a scheduler tick occurs, a long call will be made to the routine pointed to by *Vector*. The routine must do a long return when done.

If *Mode*=0, any previously requested **TimerSevice** requested by this process will be discontinued. **TimerSevice** is automatically discontinued when a task is frozen or terminated. **TimerSevice** will be reestablished when a frozen task is reawakened.

If *Mode*=1, the call arranges that the routine pointed to by DS:DX will be called as part of the clock interrupt service routine. The value returned in CX gives the approximate interval between successive clock times in milliseconds, rounded down to the nearest millisecond.

The total number of **TimerSevices** depends upon the size of a table in the system. The routine will be called at interrupt time, so it must not enable interrupts, it must return quickly, and it must not call the system.

This system call is provided so that programs can maintain a kind of real-time sense. This call entails considerable overhead, so it should be avoided if possible.