Microsoft Multitasking MS-DOS Product Specification

MEMORY MANAGEMENT

Microsoft Corporation

October 30, 1984

ABSTRACT

This memo discusses in detail the extended memory management facilities of MS-DOS 4.0. Many of these facilities are also available in the MS-Windows environment under MS-DOS 2 & 3.

1. Prerequisites

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*

Explicit and implicit reference will be made to terms and concepts introduced in these documents.

2. Introduction

The MS-DOS memory management environment:

- Provides automatic sharing of pure code segments.
- Provides maximum flexibility to relocate and/or swap segments. Swapping segments that belong solely to idle tasks is not sufficient. Segment sharing and task behavior may force swapping of some of an active task's segments, but will continue running that task until it references a swapped segment.
- Provides compatibility with the MS-DOS and XENIX operating systems.
- Is upward compatible to future systems and hardware.
- Is upward compatible (at a high level language source level) with other machine architectures (such as the 68000).
- Is efficient. The environment does not introduce significant inefficiencies in client programs.
- Is transparent. As much as possible, "ordinary" high level language programs should run without change. Likewise, minimal modifications should be necessary to allow old programs access to new services.

In summary, MS-DOS will provide a segment-based environment that creates, deletes, manages and manipulates a variety of types of segments. This task is straightforward on a 286 that has hardware relocation and protection, but it is quite complex in the 8086/8088 environment.

3. The Man Behind the Curtain

Supporting virtual segmentation in an 8086 environment is complex; whereas the 286 contains special hardware to make such features possible, the 8086 has none. In an 8086 environment, software must be used to compensate for the missing hardware.

Software is not used to emulate literally the additional 286 hardware features. The performance of such a technique would be unacceptable. The goal is to use software to provide the same results as the hardware, movable and swappable segments, but using an approach that provides acceptable performance. This requires a certain degree of cooperation from the programs that are to use these features.

Programs that exhibit such cooperation are called *conforming* programs. As will be discussed later, conformance is not an all-or-nothing situation. Programs can be partially conforming and thus be able to take advantage of a subset of the memory management features.

3.1. Moving and Swapping - Why Not?

On the 8086, the operating system can load a segment into memory wherever it wants, but once the program has begun execution it cannot be moved. This is because the program is responsible for placing the relocation values in the segment registers, and to move a segment, all the relocation values that the program might eventually load into a segment register must be found and updated. This is an impossible task.

Swapping segments without having the ability to move them is possible, but it is not useful. When it is time to swap a segment back in, it must be reloaded at its original address (because it cannot appear to have moved when the program resumes execution). This means that when a segment is swapped out to free some memory for another use, it must be guaranteed that the new user of that memory will be completed and gone before it is time to swap in the original user. This is an unmanageable restriction in a general-purpose environment where the OS has no prior knowledge about the behavior of the programs being executed.

3.2. Conforming Programs

Finding all the segment pointers in an *arbitrary* program's data area in order to move a segment is an impossible task. However, it is possible to establish programming conventions that enable the DOS to find all the segment pointers in programs that follow these conventions. To accomplish this, all pointers to a conforming program's segments are kept and used only in specific stack and register locations. If a conforming program needs to use a segment pointer in some other manner, it must first explicitly lock the segment so the DOS will not attempt to move it. When these pointers have been discarded, the program must explicitly unlock the segment.

3.3. Segmented Model

A program can be viewed as a series of segments. The different segment types are:

Fixed Segment

The segment is fixed in memory. It is used by only one process, dies when the process dies, and is never moved or swapped. Fixed segments are created by manual request or by the system when loading the automatic segments of a non-conforming program in the new-style .EXE format.

Manual Segment

The segment was explicitly created by a process. It belongs to the process, and evaporates when explicitly released or when the process dies. The segment is referenced via an indirect pointer; it may be moved or swapped.

Code Segment

A code segment may be shared by multiple tasks. It is automatically created when code is loaded, and disappears when the last user dies. It may be moved or swapped.

Auto Data – Solo Segment

An automatic data segment belonging to a code segment. All activations of the code segment share this same data segment in DS. It may be moved or swapped. It is automatically created when code is loaded; dies when code dies.

Auto Data – Instance Segment

An automatic data segment belonging to a code segment/process pair (an "instance" of the code segment). A different segment for each different process's calling of the code. It is automatically created when the instance is created; dies when that instance dies.

Physical Block

A contiguous, immovable block of memory. It is created when an old-style .EXE or .COM format program is loaded, and may occupy up to one megabyte of memory. It is similar to a fixed segment, except it is not a true segment in that it may be larger than 65K.

When a conforming program is loaded, the system creates one or more code segments and one or more data segments.

4. Manual Segments

Manual segments are explicitly created by a running process by means of an allocation call. A manual segment may be either fixed or movable; this is specified when the segment is allocated. Movable segments are manipulated and accessed via a *handle*, discussed in Section 4.1.1.

4.1. Far (Global) Memory

Far (global) segments are allocated in multiples of 16 bytes (i.e., the 8086 "paragraph" size) and have a maximum size of 64K bytes. Each far segment is aligned on a 64-byte boundary.

4.1.1. Handles

A handle is a 16-bit "name" for a segment. A handle contains sufficient information to allow a call to the memory management subsystem to locate the named segment. The low-order two bits describe the type of segment, and the remaining 14 bits contain identifier information.

To move a movable, manual segment, the system must find all pointers to it and edit them. The DOS memory manager subsystem accomplishes this by allowing only one pointer to the segment, and it stores that pointer in a memory manager data area where it can be easily located and edited. The memory manager data area also contains a count of the number of outstanding locks on the segment.

When a movable far segment is allocated manually, the DOS creates the segment, sets up the pointer, sets the lock count to zero, and returns the name for that segment: the handle. The DOS does not return the address of the segment itself; it cannot because, except while the segment is locked, there must exist only one pointer to it - the one stored in the memory management subsystem's data area.

To access a movable far segment the program must first call **LockObject()**. The **LockObject()** call locks the segment and returns a far pointer to it. Because the segment is locked, the DOS will not attempt to move it or locate the pointers to it. The program is now free to use the segment. It may reference the segment's contents, generate far pointers to items within the segment, and otherwise treat it as a fixed memory object. When the program has finished its current use of the segment, it calls **UnLockObject** to unlock the segment. After the **UnLockObject** call *all* far pointers to the segment become invalid. It is critical that the using program does not make any further reference far pointers that are generated while the segment is locked. All subsequent references to the segment must involve a new call to **LockObject** and the generation of new far pointers.

A locked segment is immovable and interferes with the DOS's allocation of memory. Consequently, **LockObject** and **UnLockObject** operate relatively quickly so that a program does not keep a movable segment locked longer than necessary. Note that a program must not remember or use the address of a segment after that segment has been unlocked; it must use **LockObject** to

redetermine the address of the segment.

4.1.2. Swapping

The memory allocator supports swapping of movable far segments. Only movable segments with a lock count of zero are eligible to be swapped. Programs have the option of identifying which segments are the best candidates for swapping; this is discussed later.

The compaction routine moves and compacts far segments in physical memory. Swapping occurs only when the compact routine is unable to acquire enough space through compaction alone. Swapping continues until the compaction procedure is able to satisfy the request for space.

When **LockObject()** is called upon a swapped-out object, it causes the system to swap the object back in before it returns to its caller. This means that, except for timing considerations, the swapping of segments is transparent to the application program.

4.2. Near (Local) Memory

Near (local) memory is allocated from the task's automatic data (DS) segment. Near objects are sized in bytes, and are aligned on 4-byte boundaries. The maximum amount of memory that can be allocated is 65K. In practice, the limit is smaller than that because of DS space consumed by static data and program stacks. The key distinction between far memory and near memory is that a far memory object is a segment and is addressed (when locked) via a 32-bit far pointer. A near memory object is a section of memory contained *within* a segment, specifically the program's automatic data segment, and it is addressed (when locked) via a 16-bit near pointer (i.e., an offset value). (Automatic data segments are discussed in more detail in Section 5.)

Like far memory, near memory can be allocated as fixed or movable, but those terms refer to being fixed or movable *with regards to their offset in the DS segment*, not with regards to the machine's physical memory. A fixed or locked near memory object will move in physical memory if the DS segment itself is moved. The object is considered fixed if the offset within that DS segment is always the same. Of course, if the programmer wishes to generate a far pointer to a near memory object he must lock (or declare fixed) both the near object and the automatic data segment itself.

Compaction is accomplished by the use of handles and the **LockObject** and **UnLockObject** subroutines. Although intentionally similar in concept, far memory and near memory differ in three important ways:

1) **LockObject** returns a far pointer to the specified object. If this object is a near object then *both* the object itself and the automatic data segment that contains it are locked. Most frequently the programmer wants to generate a near pointer to the local object, in that case **LockLocalObject** should be used. **LockLocalObject** returns a near pointer and does not lock the automatic data segment.

2) It is important to keep far memory locked for as short a time as is feasible to permit the DOS to shuffle memory freely. This restriction is relatively unimportant for near memory objects because they are moved only when the program itself requests memory; not in response to a request by some other program.

3) Near memory objects are not swapped.

Note that a fully conforming program's automatic DS segment is globally movable. In this case, an immovable near memory object has a fixed offset within that DS segment, but the segment itself may be moved in physical memory. Such moves of automatic segments are transparent to the program.

4.2.1. Handles

Alloc() returns a handle for near objects as well as for far objects. If the near object is of the *fixed* type, the two low-order bits of the handle are zero, and the handle can be used directly as the offset (near pointer) of the object. For a movable near object, the handle contains information to allow **LockObject** to locate the pointer to the object and to mark it locked until **UnLockObject** is called with that handle.

5. Automatic Segments

5.1. Definitions

Frame

A frame is a section of memory on a process's stack that contains information used by the currently executing procedure. Each process has a stack containing one or more frames. Stacks are only associated with processes, never with library routines or library segments.

Automatic Segments

An automatic segment is a segment that is implicitly allocated by the operating system to hold a program's code and data values. These segments are not explicitly allocated by the program, nor are they normally deallocated by the program. These segments are usually addressed via values kept in the CS, DS, and SS registers throughout the execution of the program. Middle-model programs may have multiple automatic code segments; only one of these segment values is in CS at any one time.

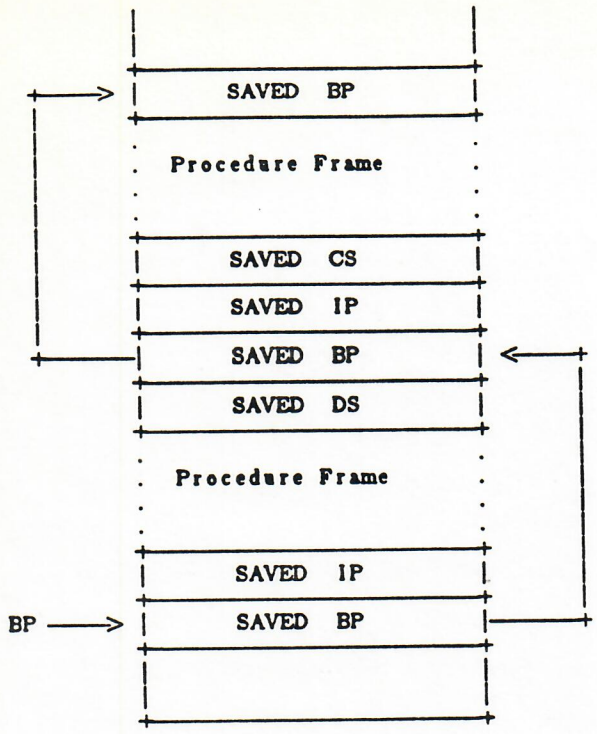
5.2. Moving Automatic Segments

Unlike manual data segments that are explicitly allocated and must be locked to be referenced, automatic data segments are automatically allocated and are automatically and transparently moved, when necessary.

An automatic data segment can be moved transparently because all references to the segment are contained in CS, SS, DS and ES, or in CS and DS images in the frame. This results in some restrictions on the use and passing of pointers to items in automatic data segments. These restrictions will be discussed in the next sections.

5.2.1. Frame Format

To move automatic code and data segments, the system must be able to "walk" the stack of the process, and to find and edit any references to the segment being moved. The format of the process's stack, the frame format, makes this possible. The required frame format is shown on the following page.



Frame Format

When a far procedure is called, CS, IP, BP and DS are saved on the stack. The low-order bit of the saved BP value is set to indicate to the DOS that this was a far call and that CS and DS were pushed on the stack.

When an automatic segment has been moved, the DOS examines the CS, DS, ES and SS registers to see if they refer to the segment. If they do, their values are edited to indicate the new location. Next, the DOS uses the BP register to locate and "walk" the stack. Each of the pushed CS and DS values is examined to see if it refers to the segment being moved. If so, the value is edited to indicate the new location.

Note that the system can only find segment references that are in the segment registers themselves or that were pushed on the stack as part of the frame format illustrated above. A segment value that is in some other register, elsewhere on the stack, or elsewhere in memory will be overlooked with disastrous consequences.

Also note that it is not permissible for conforming programs to use segment registers as "scratch pads." If the data value in a segment register happened to be equal to a segment value being moved, that data value would be changed.

5.2.2. Segment Locking

As discussed in Section 5.2., the DOS can move an automatic data segment in memory whenever it is necessary because it can check the program's segment registers and the places on the stack where the CS and DS register contents have been saved *as part of a far-procedure call.* This last statement does not mean

that the DOS can find and update *any* CS and DS values on the stack, only those that were pushed as part of a far-procedure's preamble sequence. This special sequence is generated by Microsoft's compilers; assembly language programmers should use macros to produce the code necessary to generate this format.

Sometimes it is necessary to have other occurrences of the data segment's segment number. For example, it may be necessary to generate the long-address (in C, a far-pointer, "farp") of a data segment variable to pass to some library routine. This can be done, but only *after* locking the automatic data segment so that the DOS will not try to move it. The DOS cannot tell that this extra pointer has been generated, nor can the DOS find the extra pointer. If the data segment is not explicitly locked and if the DOS moves the segment, the extra far pointer will now be pointing to some other program's memory that could cause a system crash. The rule is:

> The data segment *must* be locked *before* its segment number is put in any register (other than DS), on the stack, or in any other memory location. The data segment must *remain* locked until that far pointer is no longer being used. (The few highly-specialized exceptions to this rule are discussed in Section 5.2.4.2)

Care must be taken because if this rule is not followed, the program will *usually* run and may pass exhaustive tests, but one day it will crash the system.

For example, the following subroutine, "dataout", takes a far pointer and a byte count. This shows the correct locking procedure:

```
char static_bfr[100];

SUBR()
{
        char frame_bfr[100];
        int  ds_hand;

        ds_hand = GetDSHandle();
        LockObject(ds_hand);      /* lock for farp generation */
        dataout( (farp)static_bfr, 100);
        UnLockObject(ds_hand);

        .
        .
        LockObject(ds_hand);
        dataout( (farp)frame_bfr, 100);
        UnLockObject(ds_hand);
        .
        .
        .
```

Note that the stack is kept in the automatic data segment so it had to be locked when the *farp* of the frame address was generated. The next example shows an *incorrect* locking procedure:

```
char static_bfr[100];

SUBR()
{
        char frame_bfr[100];
        farp *fptr;
        int  ds_hand;


        ds_hand = GetDSHandle();
        fptr = (farp)static_bfr;
        LockObject(ds_hand);     /* lock for farp generation */
        dataout( fptr, 100);
        UnLockObject(ds_hand);
        .
        .
        .
```

This fragment is *incorrect* because it generates the far pointer *before* it locks the data segment. The DOS may move the data segment after the "fptr =" statement but before the **LockObject()** call.

5.2.3. Usage Guidelines - General

5.2.3.1. Locking Automatic Data Segments

A conforming program must lock its automatic data segment before any far (long) pointer is generated. Most C programs are small or middle model and they only have one data segment, so far pointers are rarely needed. (Large model programs make such extensive use of long pointers that the linker forces their data segments to be permanently fixed and thus they do not need to be explicitly locked).

The most common situation in which a small or middle-model program needs to generate a far pointer to the automatic data segment, is when it is calling a subroutine that requires a far pointer as an argument. As discussed, any movable segment must be locked before generating a far pointer to its contents. This is as true for the automatic data segment as it is for manually allocated movable segments. If the item being pointed to is a local movable memory object, its handle must also be locked so that it won't accidentally be moved within the automatic data segment.

5.2.3.2. Using Handles Instead of Segments

Note that programs written for the conforming environment need not forego a multiple-data-segment capability to retain segment movability. A conforming program can maintain a collection of addresses of items scattered among multiple

segments by saving the addresses as a *handle:offset* pair rather than a *segment:offset* pair. Routines that use these "addresses" would use the **LockObject** call to guarantee that the segment is locked before the data item is referenced. There is no problem if some of these handles represent fixed objects; **LockObject** works properly for all handle types.

Similarly, a subroutine written exclusively for the conforming environment could take a *handle:offset* pair instead of a far pointer, alleviating the need (in the calling routine) for a **LockObject** call. Naturally the subroutine itself must use **LockObject**, but it would presumably only do so immediately before referencing the object.

5.2.3.3. Minimizing Duration of Locks

The DOS's memory management performance is considerably hindered by a locked data segment so the programmer must minimize the length of time that any movable segment is kept locked. The **LockObject()** and **UnLockObject()** subroutines are fast and can be called frequently. If it appears necessary to keep a far pointer valid for a more than a short period of time, the programmer should use a different technique. For example, if the items needing "long duration" far pointers are few and/or small in size, they could be allocated in a separate, fixed segment. This would allow the program's automatic data segment, containing the program stack and the bulk of the program data, to remain movable. As a final alternative, the programmer could instruct the linker to indicate that the automatic data segment is *not* movable; it is to be permanently fixed.

If a segment is to spend most or all of its time being fixed, it is best that the DOS understand this from the beginning. The DOS stores fixed segments in a special area of memory to minimize the amount of fragmentation they cause. It's best for a segment to be movable, it's second best for it to be fixed, and it's worst of all for the segment to appear movable, be allocated in the movable area of memory, and then end up being fixed most of the time via **LockObject**.

Note:

Locking a segment for a "short time" refers to real time, not CPU time. If a program reads commands from the keyboard and if those commands are all rapid, it would be acceptable for the program to read a command, lock the data segment, execute the command, and then unlock the data segment before it reads the next command. Such a program would spend most of its time blocked on keyboard input from the **Read** call; during that time the data segment would be unlocked.

Since most interactive programs spend most of their real time blocked on command input, it is especially valuable for them to have their movable segments unlocked at those times. This is also the time that the user will most likely (via the session manager or a window package), attempt to start a new program and thus call upon the DOS for more memory.

5.2.3.4. Far Pointers to Code Segments

The DOS treats automatic code segments in a special manner because every far call into a segment contains a segment value as an operand of the instruction. This means that a great number of copies of a code segment's segment number may exist. For this reason the DOS inserts a data structure called a *call thunk* between the far call instruction and the called address. Specifically, when the DOS loads a conforming program, a call thunk is allocated in a fixed memory location known to the DOS memory manager for every far entry address. The far call instructions that call these entry points are fixed to call to the entry point's thunk, the thunk in turn jumps to the true entry point. When the DOS moves a code segment, it edits the thunk belonging to each of the segment's entry points so that they point to the new memory location.

As a result, there is no restriction upon the generation and usage of pointers to far ("global") entry points because these pointers actually point to the corresponding call thunk that is never moved. It is not necessary to lock automatic code segments in order to generate far pointers to procedures.

5.2.4. Usage Guidelines - Assembly Language

When writing a conforming assembly language program, it must be remembered that the DOS may move any automatic, movable, or unlocked segment at any time, and that the DOS will only update the segment registers and those registers stored in their proper place in the official frame format. Random disasters will occur if a program keeps segment values elsewhere without first locking the segment. Of course, this only applies to programs that declare themselves to be conforming. Programs that do not make such a declaration have their segments permanently fixed are therefore freed from these requirements. However, they will lose the benefits of being movable, swappable, 286 compatible, and being able to address greater than 640K of memory.

This section will discuss some considerations that make the writing of conforming assembly language programs sufficiently easy that all new assembly language programs should be conforming. A conforming program obeys three conventions:

1) It maintains a conforming stack and frame format
2) It uses and places values in the segment registers in the proper manner.
3) It locks and unlocks segments as necessary.

5.2.4.1. Calling Sequences

The frame format shown earlier is the "full strength" version used for far calls to a procedure. Most or all of the calls in an assembly language program are near calls and can use a simpler format. The stacks of all conforming programs have two things in common: they are word aligned and the BP register always points to the bottom-most frame. The saved BP value in that frame links upwards until a 0 value is found in the "saved BP" cell in a frame. When a task is initiated DOS will have zeroed the BP register so that a just-starting program has a legal, null-length frame chain.

5.2.4.1.1. Near Calls Without BP

The simplest form of near-call is to just use the **call** and **ret** instructions:

```
CALLER                          CALLEE

call   SUBR

                          SUBR:   .
                                  .
                                  .
                                  ret
```

This form is used when the called subroutine does not change or store the contents of SS, CS, DS, ES, or BP. In this case, BP is still pointing to an earlier valid frame; the return offset pushed by the call and any other data pushed by SUBR is seen by the DOS as just the private working storage of the frame pointed to by the BP register. Note that a small model program that never uses BP maintains a conforming frame format.

5.2.4.1.2. Near Calls With BP

If the called subroutine is to use BP to address its frame then it must save BP in the proper manner so that DOS can still traverse the stack. The sequence is:

```
CALLER                          CALLEE

call   SUBR

                   SUBR:  push    BP
                          mov     BP,SP
                          .
                          .                    ; BP unchanged
                          .
                          mov     SP,BP
                          pop     BP
                          ret
```

Since the pushed BP value was EVEN when the DOS scanned up the stack, no CS or DS value was saved at this spot.

5.2.4.1.3. Far Calls Without DS

The far call sequence differs from the near call sequence in two ways. First, the BP value is incremented before it is pushed. This alerts the DOS to the fact that this frame contains saved CS and DS values. Second, the DS register must be pushed. This is done even in the case shown where the called routine will not change the contents of DS. This is necessary because if it were omitted, the DOS might erroneously edit the first data value pushed after BP in the mistaken belief that it represents a segment value.

As previously noted, the far-call instruction contains a segment specification within it. If the called segment is movable the DOS causes the **call** instruction

to point to a long-jump instruction. The memory manager routines maintain this jump instruction so that it points to the movable segment at all times. This insertion of a "jump thunk" is done at load time and is transparent to the programmer.

CALLER

```
call    SUBR    ; far call
```

CALLEE

```
SUBR:   inc     bp
        push    bp
        mov     bp,sp
        push    ds
        .
        .                       ; BP unchanged
        .
        mov     sp,bp
        pop     bp
        dec     bp
        ret                     ; far return
```

5.2.4.1.4. Far Calls With DS

This form of the far call sequence is used for subroutines that have their own automatic data segments. This situation is not common, it comes up when using certain forms of dynamically linked subroutines. Refer to the *Microsoft Multitasking MS-DOS Product Specification DYNAMIC LINKING* document for details. As in the previous example, the DOS inserts a jump thunk. In addition, the thunk loads AX with the called subroutine's automatic DS value, so AX cannot be preserved when calling "external segments with instance data segments."

CALLER

```
call    SUBR    ; far call
```

CALLEE

```
SUBR:   inc     bp
        push    bp
        mov     bp,sp
        push    ds
        mov     ds,ax
        .                       ; BP unchanged
        .
        .
        sub     bp,2
        mov     sp,bp
        pop     ds
        pop     bp
        dec     bp
        ret                     ; far return
```

5.2.4.2. Segment Register Manipulation

The preceding text emphasizes that an assembly language program must not manipulate the contents of CS, SS, DS, ES, and BP. This was done for the purpose of simplification; in actuality a program *may* manipulate the contents of DS and ES if the proper conventions are observed.

SS must never be changed; the DOS uses SS:BP to traverse the stack and it assumes that the stack is wholly contained within the SS segment. CS may only be changed via far calls, far jumps, and far returns. These can only be done in the exact form shown in the previous example. It is *not* sufficient to substitute an "equivalent" code sequence, because the DOS may move a segment while the program is in the process of executing the frame setup/takedown sequences. At these times the frame format is invalid because it is incomplete. Immediately before scanning the stack, the DOS examines the next instructions to be executed. If they are part of the code sequences shown, the DOS understands that the frame information is incomplete and handles the situation properly. *The use of a different instruction sequence to build or tear down the frame, or the use of those specific instruction sequences elsewhere than subroutine entry and exit is forbidden.*

The restrictions on the use of DS and ES are more flexible. Two different cases will be considered: the loading of DS or ES with the segment number of a fixed or locked segment, and the loading of DS or ES with the segment number of an automatic code or data segment.

- Loading DS/ES With a Fixed or Locked Segment

  Since there are no restrictions on the use of a fixed or locked segment's segment number (or on the creation or use of a far pointer to an item in such a segment), the programmer can use any method desired to load the value into DS or ES. However, a problem *does* arise if the programmer wants to save the previous contents of those registers. If a segment register contains the segment number of an unlocked automatic segment, then its contents may not be stored or pushed. To do so would create an unfindable (and therefore unfixable) reference to the segment.

  Most of the time the previous DS or ES value is that of the automatic data segment. This value is usually also in SS, so the programmer can avoid the problem by reloading DS or ES from SS, instead of by saving and restoring it.

- Loading DS/ES With an Automatic Segment Number

  The loading of DS/ES with an automatic segment number presents some special problems. Once the value is in DS or ES, then everything is fine. The problem is, that since there are no "move segreg-to-segreg" instructions, the automatic segment value must be staged through a memory location or a general-purpose register. The presence of an unlocked automatic segment number in a general register or memory cell is illegal, even if only for a very short time. A correct interpretation of the DOS segment register

restrictions will show that the sequence:

```
PUSH   SS
POP    DS
```

is improper, because the automatic data segment's number is momentarily present on the stack.

The fact that this sequence is forbidden is burdensome to the assembly language programmer, so the DOS has special provisions to recognize and properly handle the following code sequences:

```
PUSH   SS
POP    DS

PUSH   SS
POP    ES

PUSH   CS
POP    DS

PUSH   CS
POP    DS

PUSH   DS
POP    ES

PUSH   ES
POP    DS
```

Note that the sequences that load the CS value into DS or ES should only be used to read data items from the CS segment. Programs that write into the CS segment are *not* 286 compatible.

5.2.4.3. Disabling Interrupts

There exists one additional technique that the assembly language programmer can use when manipulating segment registers: temporarily disabling interrupts. Since the DOS cannot take control away from the program while interrupts are disabled, a program can:

Disable interrupts.
Do something non-conforming with segment register values.
Become conforming again.
Enable interrupts.

Note: Interrupts may be disabled only for a very short time; no more than 100 microseconds. The disabling of interrupts for a longer interval will prevent the DOS from servicing interrupts and scheduling tasks at its specified rate. This can can cause network and communications programs to lose data and fail.

5.2.4.4. System Calls

Many system calls require an address to be passed in a segment-register offset-register pair, typically DS:DX. The DOS is written so that if the DS segment is moved during the system call the system call will proceed properly. Upon exit from the system call the DS register will contain the new value.

6. Advanced Features

6.1. Swap Advisory

When a far memory object is allocated, a flag bit can be set to indicate whether that object is a good candidate for swapping; that is, the information that it contains will probably not be used for a while. However, the setting or clearing of this bit does not guarantee or prevent the swapping of the object. This bit is strictly advisory.

6.2. Discard

Memory segments can be marked discardable and given a urgency level. The system defines "urgency" levels (shown below) as a range from 0 to 15, in increasing levels of importance. A segment marked "discard level 2" will be discarded when level 0 and level 1 actions are not yielding the memory needed.

At each urgency level the DOS takes the appropriate action for that level. If this does not free the necessary space, the DOS advances to the next level.

The applicable urgency levels are:

Level 2:

Data that can be easily regenerated.

Level 3:

Data that requires some CPU time to regenerate.

Level 5:

Data that requires some I/O to regenerate.

Level 7:

The DOS discards data that requires significant I/O to regenerate.

Appendix A

Subroutine Calls

1. Handles

The handle is a 16-bit pointer to an object that contains sufficient information to allow a client program to access the object. The low-order bit (bit 0) of a handle is zero for local objects and one for global objects. The next low-order bit of a handle (bit 1) is zero for fixed objects and one for movable objects. The remaining 14 bits (bits 2-15) contain a pointer. Local objects are allocated in bytes. Global objects are allocated in blocks of 16 bytes (paragraphs). Since the low-order two bits of a handle are used to encode type information about the object to which it points, this forces certain alignment restrictions. For local objects, it means that they must be aligned on 4-byte boundaries. For global objects, it means they must be aligned on 64-byte boundaries. The two type bits are the minimum information needed by the memory allocator to dereference a handle.

The 14 pointer bits in a handle are interpreted as follows:

Fixed, local object

Pointer is an offset within the containing global object that the local object was allocated in. Since the low-order two bits are zero, this pointer can be used directly access the contents of the object, provided a segment register contains the address of the global object that contains the local object.

Movable, local object

Pointer is an offset within the containing global object that the local object was allocated in. The pointer points to a two-word block. The first word contains the count of the number of outstanding copies of the pointer (i.e., a resource semaphore). The second word contains the actual pointer to the object.

Fixed, global object

Pointer is a "paragraph" address of an object in physical memory. The object can be accessed by masking out the low-order bit, and loading the result into a segment register.

Movable, global object

Pointer is an offset within the memory allocator's data segment of a two-word block of memory. The first word contains the count of the number of outstanding copies of the pointer (i.e., it is a resource semaphore). The second word contains the actual "paragraph" address of the object in physical memory.

2. Calls

Alloc (*Flags, Size*)

Purpose

This routine allocates an object from memory of the type specified.

Parameters

Flags defines several flags that determine how the object will be allocated as described below. Note that flags other than global/local and fixed/movable are subject to change at any time. All other bits are reserved for Microsoft use and must be zero.

local/global flag

Determines whether the object will be allocated from the caller's data segment (local) or from physical memory (global).

fixed/movable flag

Determines whether the object is movable or not. If fixed, the returned handle points directly to the allocated object.

discard flag

If set indicates that the segment is to be discarded when the system runs out of memory, rather than swapped. This bit may be set and cleared dynamically via the **ReAlloc** call.

swap candidate flag

If set indicates that the segment is infrequently used and is a good candidate for swapping *if the DOS finds that it must swap.* The setting of this flag does not guarantee that a segment will be swapped; nor does the absence of this bit guarantee that a segment will *not* be swapped.

Size is the total number of bytes to allocate. The actual number of bytes allocated may be larger due to alignment constraints.

Result

If successful, this routine returns the handle of the requested segment, otherwise, it returns zero.

LocalAlloc (*Flags, nBytes*)

Purpose

This routine is a simpler version of **Alloc** used to allocate local objects only.

Parameters

Flags defines several flags that determine how the object will be allocated as described below. All undefined bits are reserved for Microsoft use and must be zero.

fixed/movable flag

Determines whether the object is movable or not. If fixed, the returned handle points directly to the allocated object.

nBytes is the total number of bytes to allocate. The actual number of bytes allocated may be larger due to alignment constraints.

Result

If successful, this routine returns the handle of the memory object. Otherwise, it returns zero.

ReAlloc (*Handle, Size, FlagMask, FlagValues*)

Purpose

This routine changes the allocated size and/or flags of an object.

Parameters

Handle identifies the object to be changed.

Size specifies the new size of the object.

FlagMask and *FlagValues* are used to optionally change the setting of some of the object descriptor flags. The value of every flag specified in *flagMask* is changed to the value of the flag set in *FlagValues*. A zero value for *FlagMask* indicates no change in flag state. The only flags that may be changed are:

swap candidate flag

keep/discard flag

Result

If successful, this routine returns the handle to the reallocated object; otherwise, it returns zero.

Notes

A fixed memory object may not be extendible; there may be another fixed object adjacent to it in memory. If so, the user may want to allocate a new fixed area of the larger size, copy the contents of the old area, and then release the old area. In effect, this involves the moving of a fixed segment, so the DOS will not perform this process automatically. The movement must be done by the application program to guarantee the appropriateness of the procedure.

Commonly the DOS may be unable to extend the size of movable segments that are currently locked. It is strongly recommended that all movable segments be unlocked before their size is extended.

LocalReAlloc (*Handle, Size*)

Purpose

This routine is a high speed, simpler version of ReAlloc used to allocate local objects only.

Parameters

Handle identifies the object to be changed.

Size specifies the new size of the object.

Result

If successful, this routine returns the handle of the reallocated object; otherwise, it returns zero.

GetDSHandle ()

Purpose

This routine returns the handle of the calling process's automatic data segment. Since this value is a handle it will remain valid throughout the execution of the program. This handle can be used to lock and unlock the automatic data segment.

GetSize (*Handle*)

Purpose

This routine returns the current size of the object specified by *Handle*.

Parameters

Handle identifies the object.

Result

If successful, this routine returns the size of the object specified by *Handle*. If an invalid object is specified, it returns zero.

LockObject (*Handle*)

Purpose

This routine dereferences the handle of the object identified by *Handle* into a physical address. If *Handle* specifies a local movable object then both the object itself and the automatic data segment containing it are locked.

Parameters

Handle identifies the object to be dereferenced.

Result

If successful, this routine returns a far pointer to the object specified. If an invalid object is specified, it returns zero.

Notes

For movable objects, this routine increments the lock count in the handle table entry.

For fixed objects, this routine does nothing, except convert the passed handle into a physical address by masking out the low-order two bits.

Invocations of **LockObject** and **LockLocalObject** cause objects to be inswapped.

LockLocalObject (*Handle*)

Purpose

This routine is a version of **LockObject** used to dereference local objects only. Its use is advantageous because it returns a 16-bit near pointer ("offset") and does not cause the automatic data segment itself to be locked.

Parameters

Handle identifies the object to be dereferenced.

Result

If successful, this routine returns a 16-bit near pointer. If an invalid object is specified, it returns zero.

UnLockObject (*Handle*)

Purpose

This routine decrements the lock count for the object specified by *Handle*, undoing the effect of a **LockObject** call. The programmer must insure that no far pointers obtained while the object was locked are every used again. Future references to the segment must be made by relocking the segment *and obtaining new far pointers* to objects within it.

Parameters

Handle identifies the object.

Result

If successful, this routine returns a Boolean value of TRUE. If the lock count of the object was already zero, it returns FALSE.

Notes

If **LockObject** is used to lock a local object, it is important that **UnLockObject** be used correspondingly. If **UnLockLocalObject** is used, then the automatic data segment will be left locked.

UnLockLocalObject (*Handle*)

Purpose

This routine decrements the lock count for the local object specified by *Handle*.

Parameters

Handle identifies the local object.

Result

If successful, this routine returns a Boolean value of TRUE. If the lock count of the object was already zero, it returns FALSE.

Notes

If **LockObject** is used to lock a local object it is important that **UnLockObject** be used correspondingly. If **UnLockLocalObject** is used then the automatic data segment would be left locked.

Free (*Handle*)

Purpose

This routine destroys the object identified by *Handle* and returns its memory to the system.

Parameters

Handle identifies the object.

Result

If successful, this routine returns zero. If unsuccessful, it returns the *Handle*.

IsSwapped (*Handle*)

Purpose

This routine determines whether the global object identified by *Handle* is swapped in or out of memory.

Parameters

Handle identifies the global object.

Result

If the global object identified by *Handle* is currently swapped out to secondary storage, this routine returns a Boolean value of TRUE.