Article 4
Structure of an Application Program

Planning an MS-DOS application program requires serious analysis of the program's size. This analysis can help the programmer determine which of the two program styles supported by MS-DOS best suits the application. The .EXE program structure provides a large program with benefits resulting from the extra 512 bytes (or more) of header that preface all .EXE files. On the other hand, at the cost of losing the extra benefits, the .COM program structure does not burden a small program with the overhead of these extra header bytes.

Because .COM programs start their lives as .EXE programs (before being converted by EXE2BIN) and because several aspects of application programming under MS-DOS remain similar regardless of the program structure used, a solid understanding of .EXE structures is beneficial even to the programmer who plans on writing only .COM programs. Therefore, we'll begin our discussion with the structure and behavior of .EXE programs and then look at differences between .COM programs and .EXE programs, including restrictions on the structure and content of .COM programs.

The .EXE Program

The .EXE program has several advantages over the .COM program for application design. Considerations that could lead to the choice of the .EXE format include

- Extremely large programs
- Multiple segments
- Overlays
- Segment and far address constants
- Long calls
- Possibility of upgrading programs to MS OS/2 protected mode

The principal advantages of the .EXE format are provided by the file header. Most important, the header contains information that permits a program to make direct segment address references—a requirement if the program is to grow beyond 64 KB.

The file header also tells MS-DOS how much memory the program requires. This information keeps memory not required by the program from being allocated to the program—an important consideration if the program is to be upgraded in the future to run efficiently under MS OS/2 protected mode.

Before discussing the .EXE program structure in detail, we'll look at how .EXE programs behave.

Giving control to the .EXE program

Figure 4-1 gives an example of how a .EXE program might appear in memory when MS-DOS first gives the program control. The diagram shows Microsoft's preferred program segment arrangement.

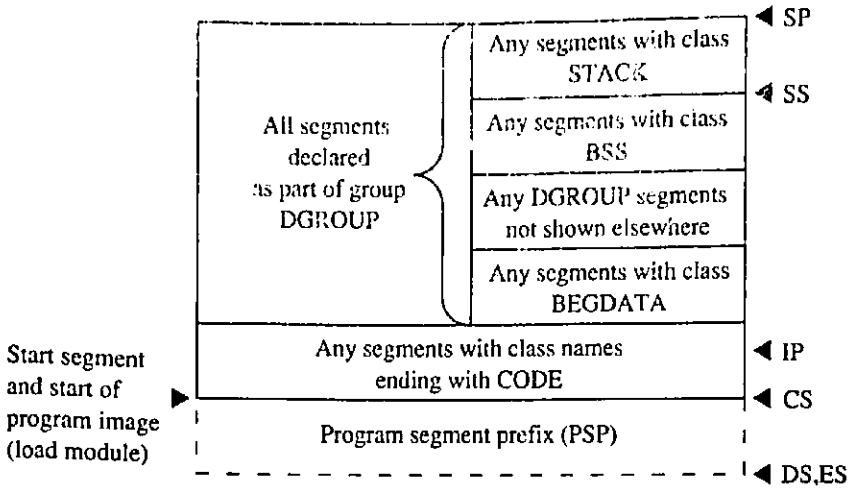

Figure 4-1. The .EXE program: memory map diagram with register pointers.

Before transferring control to the .EXE program, MS-DOS initializes various areas of memory and several of the microprocessor's registers. The following discussion explains what to expect from MS-DOS before it gives the .EXE program control.

The program segment prefix

The program segment prefix (PSP) is not a direct result of any program code. Rather, this special 256-byte (16-paragraph) page of memory is built by MS-DOS in front of all .EXE and .COM programs when they are loaded into memory. Although the PSP does contain several fields of use to newer programs, it exists primarily as a remnant of CP/M—Microsoft adopted the PSP for ease in porting the vast number of programs available under CP/M to the MS-DOS environment. Figure 4-2 shows the fields that make up the PSP.

PSP:0000H (Terminate [old Warm Boot] Vector) The PSP begins with an 8086-family INT 20H instruction, which the program can use to transfer control back to MS-DOS. The PSP includes this instruction at offset 00H because this address was the WBOOT (Warm Boot/Terminate) vector under CP/M and CP/M programs usually terminated by jumping to this vector. This method of termination should not be used in newer programs. *See* Terminating the .EXE Program below.

PSP:0002H (Address of Last Segment Allocated to Program) MS-DOS introduced the word at offset 02H into the PSP. It contains the segment address of the paragraph following the block of memory allocated to the program. This address should be used only to determine the size or the end of the memory block allocated to the program; it must not be considered a pointer to free memory that the program can appropriate. In most cases this address will *not* point to free memory, because any free memory will already have been

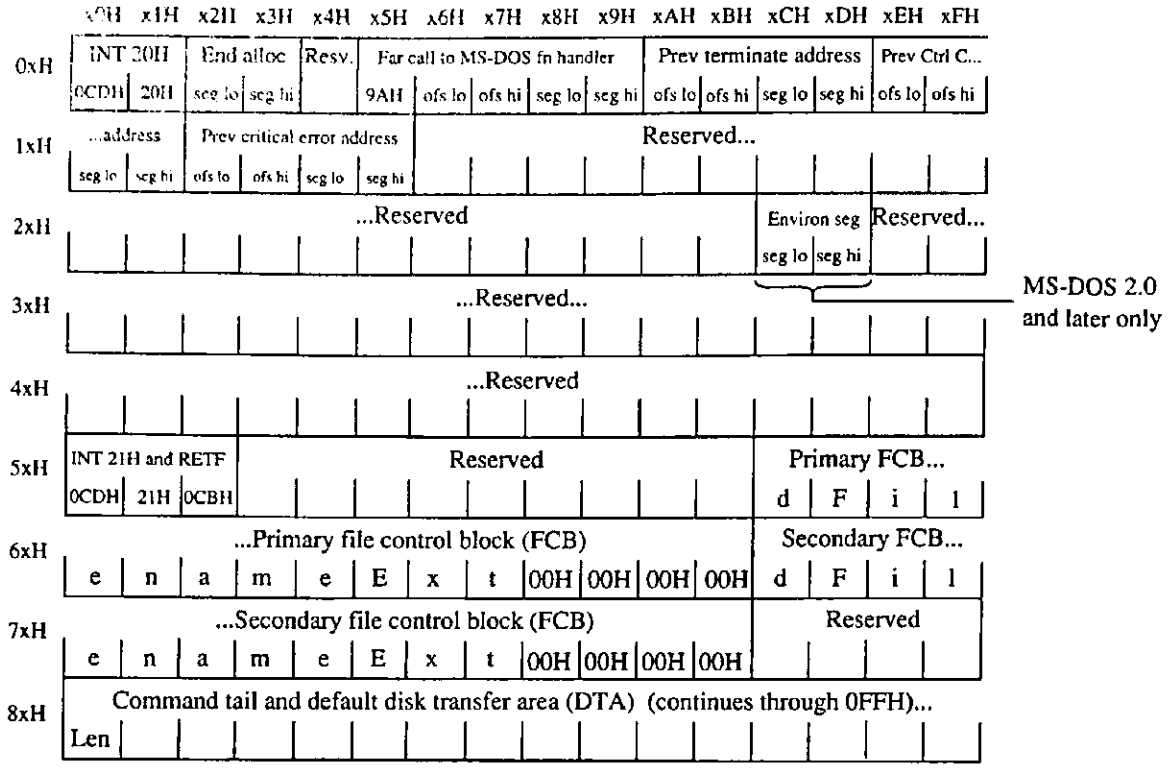

Figure 4-2. The program segment prefix (PSP).

allocated to the program unless the program was linked using the /CPARMAXALLOC switch. Even when /CPARMAXALLOC is used, MS-DOS may fit the program into a block of memory only as big as the program requires. Well-behaved programs should acquire additional memory only through the MS-DOS function calls provided for that purpose.

PSP:0005H (MS-DOS Function Call [old BDOS] Vector) Offset 05H is also a hand-me-down from CP/M. This location contains an 8086-family far (intersegment) call instruction to MS-DOS's function request handler. (Under CP/M, this address was the Basic Disk Operating System [BDOS] vector, which served a similar purpose.) This vector should not be used to call MS-DOS in newer programs. The System Calls section of this book explains the newer, approved method for calling MS-DOS. MS-DOS provides this vector only to support CP/M-style programs and therefore honors only the CP/M-style functions (00–24H) through it.

PSP:000AH-0015H (Parent's 22H, 23H, and 24H Interrupt Vector Save) MS-DOS uses offsets 0AH through 15H to save the contents of three program-specific interrupt vectors. MS-DOS must save these vectors because it permits any program to execute another program (called a child process) through an MS-DOS function call that returns control to the original program when the called program terminates. Because the original program resumes executing when the child program terminates, MS-DOS must restore these three

interrupt vectors for the original program in case the called program changed them. The three vectors involved include the program termination handler vector (Interrupt 22H), the Control-C/Control-Break handler vector (Interrupt 23H), and the critical error handler vector (Interrupt 24H). MS-DOS saves the original preexecution contents of these vectors in the child program's PSP as doubleword fields beginning at offsets 0AH for the program termination handler vector, 0EH for the Control-C/Control-Break handler vector, and 12H for the critical error handler vector.

PSP:002CH (Segment Address of Environment) Under MS-DOS versions 2.0 and later, the word at offset 2CH contains one of the most useful pieces of information a program can find in the PSP—the segment address of the first paragraph of the MS-DOS environment. This pointer enables the program to search through the environment for any configuration or directory search path strings placed there by users with the SET command.

PSP:0050H (New MS-DOS Call Vector) Many programmers disregard the contents of offset 50H. The location consists simply of an INT 21H instruction followed by a RETF. A .EXE program can call this location using a far call as a means of accessing the MS-DOS function handler. Of course, the program can also simply do an INT 21H directly, which is smaller and faster than calling 50H. Unlike calls to offset 05H, calls to offset 50H can request the full range of MS-DOS functions.

PSP:005CH (Default File Control Block 1) and PSP:006CH (Default File Control Block 2) MS-DOS parses the first two parameters the user enters in the command line following the program's name. If the first parameter qualifies as a valid (limited) MS-DOS filename (the name can be preceded by a drive letter but not a directory path), MS-DOS initializes offsets 5CH through 6BH with the first 16 bytes of an unopened file control block (FCB) for the specified file. If the second parameter also qualifies as a valid MS-DOS filename, MS-DOS initializes offsets 6CH through 7BH with the first 16 bytes of an unopened FCB for the second specified file. If the user specifies a directory path as part of either filename, MS-DOS initializes only the drive code in the associated FCB. Many programmers no longer use this feature, because file access using FCBs does not support directory paths and other newer MS-DOS features.

Because FCBs expand to 37 bytes when the file is opened, opening the first FCB at offset 5CH causes it to grow from 16 bytes to 37 bytes and to overwrite the second FCB. Similarly, opening the second FCB at offset 6CH causes it to expand and to overwrite the first part of the command tail and default disk transfer area (DTA). (The command tail and default DTA are described below.) To use the contents of both default FCBs, the program should copy the FCBs to a pair of 37-byte fields located in the program's data area. The program can use the first FCB without moving it only after relocating the second FCB (if necessary) and only by performing sequential reads or writes when using the first FCB. To perform random reads and writes using the first FCB, the programmer must either move the first FCB or change the default DTA address. Otherwise, the first FCB's random record field will overlap the start of the default DTA. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: PROGRAMMING FOR MS-DOS: File and Record Management.

PSP:0080H (Command Tail and Default DTA) The default DTA resides in the entire second half (128 bytes) of the PSP. MS-DOS uses this area of memory as the default record buffer if the program uses the FCB-style file access functions. Again, MS-DOS inherited this location from CP/M. (MS-DOS provides a function the program can call to change the address MS-DOS will use as the current DTA. *See* SYSTEM CALLS: INTERRUPT 21H: Function 1AH.) Because the default DTA serves no purpose until the program performs some file activity that requires it, MS-DOS places the command tail in this area for the program to examine. The command tail consists of any text the user types following the program name when executing the program. Normally, an ASCII space (20H) is the first character in the command tail, but any character MS-DOS recognizes as a separator can occupy this position. MS-DOS stores the command-tail text starting at offset 81H and always places an ASCII carriage return (0DH) at the end of the text. As an additional aid, it places the length of the command tail at offset 80H. This length includes all characters except the final 0DH. For example, the command line

```
C>DOIT WITH CLASS  <Enter>
```

will result in the program DOIT being executed with PSP:0080H containing

```
0B  20 57 49 54 48 20 43 4C 41 53 53 0D
len sp W  I  T  H  sp C  L  A  S  S  cr
```

The stack

Because .EXE-style programs did not exist under CP/M, MS-DOS expects .EXE programs to operate in strictly MS-DOS fashion. For example, MS-DOS expects the .EXE program to supply its own stack. (Figure 4-1 shows the program's stack as the top box in the diagram.)

Microsoft's high-level-language compilers create a stack themselves, but when writing in assembly language the programmer must specifically declare one or more segments with the STACK *combine* type. If the programmer declares multiple stack segments, possibly in different source modules, the linker combines them into one large segment. *See* Controlling the .EXE Program's Structure below.

Many programmers declare their stack segments as preinitialized with some recognizable repeating string such as **STACK*. This makes it possible to examine the program's stack in memory (using a debugger such as DEBUG) to determine how much stack space the program actually used. On the other hand, if the stack is left as uninitialized memory and linked at the end of the .EXE program, it will not require space within the .EXE file. (The reason for this will become more apparent when we examine the structure of a .EXE file.)

Note: When multiple stack segments have been declared in different .ASM files, the Microsoft Object Linker (LINK) correctly allocates the total amount of stack space specified in all the source modules, but the initialization data from all modules is overlapped module by module at the high end of the combined segment.

An important difference between .COM and .EXE programs is that MS-DOS preinitializes a .COM program's stack with a termination address before transferring control to the program. MS-DOS does not do this for .EXE programs, so a .EXE program *cannot* simply execute an 8086-family RET instruction as a means of terminating.

Note: In the assembly-language files generated for a Microsoft C program or for programs in most other high-level-languages, the compiler's placement of a RET instruction at the end of the *main* function/subroutine/procedure might seem confusing. After all, MS-DOS does not place any return address on the stack. The compiler places the RET at the end of *main* because *main* does not receive control directly from MS-DOS. A library initialization routine receives control from MS-DOS; this routine then calls *main*. When *main* performs the RET, it returns control to a library termination routine, which then terminates back to MS-DOS in an approved manner.

Preallocated memory

While loading a .EXE program, MS-DOS performs several steps to determine the initial amount of memory to be allocated to the program. First, MS-DOS reads the two values the linker places near the start of the .EXE header: The first value, MINALLOC, indicates the minimum amount of extra memory the program requires to start executing; the second value, MAXALLOC, indicates the maximum amount of extra memory the program would like allocated before it starts executing. Next, MS-DOS locates the largest free block of memory available. If the size of the program's image within the .EXE file combined with the value specified for MINALLOC exceeds the memory block it found, MS-DOS returns an error to the process trying to load the program. If that process is COMMAND.COM, COMMAND.COM then displays a *Program too big to fit in memory* error message and terminates the user's execution request. If the block exceeds the program's MINALLOC requirement, MS-DOS then compares the memory block against the program's image combined with the MAXALLOC request. If the free block exceeds the maximum memory requested by the program, MS-DOS allocates only the maximum request; otherwise, it allocates the entire block. MS-DOS then builds a PSP at the start of this block and loads the program's image from the .EXE file into memory following the PSP.

This process ensures that the extra memory allocated to the program will immediately follow the program's image. The same will not necessarily be true for any memory MS-DOS allocates to the program as a result of MS-DOS function calls the program performs during its execution. Only function calls requesting MS-DOS to increase the initial allocation can guarantee additional contiguous memory. (Of course, the granting of such increase requests depends on the availability of free memory following the initial allocation.)

Programmers writing .EXE programs sometimes find the lack of keywords or compiler/assembler switches that deal with MINALLOC (and possibly MAXALLOC) confusing. The programmer never explicitly specifies a MINALLOC value because LINK sets MINALLOC to the total size of all uninitialized data and/or stack segments linked at the very end of the program. The MINALLOC field allows the compiler to indicate the size of the initialized data fields in the load module without actually including the fields themselves, resulting in a smaller .EXE program file. For LINK to minimize the size of the .EXE file, the program must be coded and linked in such a way as to place all uninitialized data fields at the end of the program. Microsoft high-level-language compilers handle this automatically; assembly-language programmers must give LINK a little help.

Note: Beginning and even advanced assembly-language programmers can easily fall into an argument with the assembler over field addressing when attempting to place data fields after the code in the source file. This argument can be avoided if programmers use the SEGMENT and GROUP assembler directives. *See* Controlling the .EXE Program's Structure below.

No reliable method exists for the linker to determine the correct MAXALLOC value required by the .EXE program. Therefore, LINK uses a "safe" value of FFFFH, which causes MS-DOS to allocate all of the largest block of free memory—which is usually *all* free memory—to the program. Unless a program specifically releases the memory for which it has no use, it denies multitasking supervisor programs, such as IBM's TopView, any memory in which to execute additional programs—hence the rule that a well-behaved program releases unneeded memory during its initialization. Unfortunately, this memory conservation approach provides no help if a multitasking supervisor supports the ability to load several programs into memory without executing them. Therefore, programs that have correctly established MAXALLOC values actually are well-behaved programs.

To this end, newer versions of Microsoft LINK include the /CPARMAXALLOC switch to permit specification of the maximum amount of memory required by the program. The /CPARMAXALLOC switch can also be used to set MAXALLOC to a value that is known to be less than MINALLOC. For example, specifying a MAXALLOC value of 1 (/CP:1) forces MS-DOS to allocate only MINALLOC extra paragraphs to the program. In addition, Microsoft supplies a program called EXEMOD with most of its languages. This program permits modification of the MAXALLOC field in the headers of existing .EXE programs. *See* Modifying the .EXE File Header below.

The registers

Figure 4-1 gives a general indication of how MS-DOS sets the 8086-family registers before transferring control to a .EXE program. MS-DOS determines most of the original register values from information the linker places in the .EXE file header at the start of the .EXE file.

MS-DOS sets the SS register to the segment (paragraph) address of the start of any segments declared with the STACK *combine* type and sets the SP register to the offset from SS of the byte immediately after the combined stack segments. (If no stack segment is declared, MS-DOS sets SS:SP to CS:0000.) Because in the 8086-family architecture a stack grows from high to low memory addresses, this effectively sets SS:SP to point to the base of the stack. Therefore, if the programmer declares stack segments when writing an assembly-language program, the program will not need to initialize the SS and SP registers. Microsoft's high-level-language compilers handle the creation of stack segments automatically. In both cases, the linker determines the initial SS and SP values and places them in the header at the start of the .EXE program file.

Unlike its handling of the SS and SP registers, MS-DOS does *not* initialize the DS and ES registers to any data areas of the .EXE program. Instead, it points DS and ES to the start of

the PSP. It does this for two primary reasons: First, MS-DOS uses the DS and ES registers to tell the program the address of the PSP; second, most programs start by examining the command tail within the PSP. Because the program starts without DS pointing to the data segments, the program must initialize DS and (optionally) ES to point to the data segments before it starts trying to access any fields in those segments. Unlike .COM programs, .EXE programs can do this easily because they can make direct references to segments, as follows:

```
MOV     AX,SEG DATA_SEGMENT_OR_GROUP_NAME
MOV     DS,AX
MOV     ES,AX
```

High-level-language programs need not initialize and maintain DS and ES; the compiler and library support routines do this.

In addition to pointing DS and ES to the PSP, MS-DOS also sets AH and AL to reflect the validity of the drive identifiers it placed in the two FCBs contained in the PSP. MS-DOS sets AL to 0FFH if the first FCB at PSP:005CH was initialized with a nonexistent drive identifier; otherwise, it sets AL to zero. Similarly, MS-DOS sets AH to reflect the drive identifier placed in the second FCB at PSP:006CH.

When MS-DOS analyzes the first two command-line parameters following the program name in order to build the first and second FCBs, it treats *any* character followed by a colon as a drive prefix. If the drive prefix consists of a lowercase letter (ASCII *a* through *z*), MS-DOS starts by converting the character to uppercase (ASCII *A* through *Z*). Then it subtracts 40H from the character, regardless of its original value. This converts the drive prefix letters A through Z to the drive codes 01H through 1AH, as required by the two FCBs. Finally, MS-DOS places the drive code in the appropriate FCB.

This process does not actually preclude invalid drive specifications from being placed in the FCBs. For instance, MS-DOS will accept the drive prefix !: and place a drive code of 0E1H in the FCB (! = 21H; 21H−40H = 0E1H). However, MS-DOS will then check the drive code to see if it represents an existing drive attached to the computer and will pass a value of 0FFH to the program in the appropriate register (AL or AH) if it does not.

As a side effect of this process, MS-DOS accepts @: as a valid drive prefix because the subtraction of 40H converts the @ character (40H) to 00H. MS-DOS accepts the 00H value as valid because a 00H drive code represents the current default drive. MS-DOS will leave the FCB's drive code set to 00H rather than translating it to the code for the default drive because the MS-DOS function calls that use FCBs accept the 00H code.

Finally, MS-DOS initializes the CS and IP registers, transferring control to the program's entry point. Programs developed using high-level-language compilers usually receive control at a library initialization routine. A programmer writing an assembly-language program using the Microsoft Macro Assembler (MASM) can declare any label within the

program as the entry point by placing the label after the END statement as the last line of the program:

```
END     ENTRY_POINT_LABEL
```

With multiple source files, only one of the files should have a label following the END statement. If more than one source file has such a label, LINK uses the first one it encounters as the entry point.

The other processor registers (BX, CX, DX, BP, SI, and DI) contain unknown values when the program receives control from MS-DOS. Once again, high-level-language programmers can ignore this fact—the compiler and library support routines deal with the situation. However, assembly-language programmers should keep this fact in mind. It may give needed insight sometime in the future when a program functions at certain times and not at others.

In many cases, debuggers such as DEBUG and SYMDEB initialize uninitialized registers to some predictable but undocumented state. For instance, some debuggers may predictably set BP to zero before starting program execution. However, a program must not rely on such debugger actions, because MS-DOS makes no such promises. Situations like this could account for a program that fails when executed directly under MS-DOS but works fine when executed using a debugger.

Terminating the .EXE program

After MS-DOS has given the .EXE program control and it has completed whatever task it set out to perform, the program needs to give control back to MS-DOS. Because of MS-DOS's evolution, five methods of program termination have accumulated—not including the several ways MS-DOS allows programs to terminate but remain resident in memory.

Before using any of the termination methods supported by MS-DOS, the program should always close any files it had open, especially those to which data has been written or whose lengths were changed. Under versions 2.0 and later, MS-DOS closes any files opened using handles. However, good programming practice dictates that the program not rely on the operating system to close the program's files. In addition, programs written to use shared files under MS-DOS versions 3.0 and later should release any file locks before closing the files and terminating.

The Terminate Process with Return Code function

Of the five ways a program can terminate, only the Interrupt 21H Terminate Process with Return Code function (4CH) is recommended for programs running under MS-DOS version 2.0 or later. This method is one of the easiest approaches to terminating *any* program, regardless of its structure or segment register settings. The Terminate Process with Return Code function call simply consists of the following:

```
MOV     AH,4CH                  ;load the MS-DOS function code
MOV     AL,RETURN_CODE          ;load the termination code
INT     21H                     ;call MS-DOS to terminate program
```

The example loads the AH register with the Terminate Process with Return Code function code. Then it loads the AL register with a return code. Normally, the return code represents the reason the program terminated or the result of any operation the program performed.

A program that executes another program as a child process can recover and analyze the child program's return code if the child process used this termination method. Likewise, the child process can recover the RETURN_CODE returned by any program it executes as a child process. When a program is terminated using this method and control returns to MS-DOS, a batch (.BAT) file can be used to test the terminated program's return code using the *IF ERRORLEVEL* statement.

Only two general conventions have been adopted for the value of RETURN_CODE: First, a RETURN_CODE value of 00H indicates a normal no-error termination of the program; second, increasing RETURN_CODE values indicate increasing severity of conditions under which the program terminated. For instance, a compiler could use the RETURN_CODE 00H if it found no errors in the source file, 01H if it found only warning errors, or 02H if it found severe errors.

If a program has no need to return any special RETURN_CODE values, then the following instructions will suffice to terminate the program with a RETURN_CODE of 00H:

```
        MOV     AX,4C00H
        INT     21H
```

Apart from being the approved termination method, Terminate Process with Return Code is easier to use with .EXE programs than any other termination method because all other methods require that the CS register point to the start of the PSP when the program terminates. This restriction causes problems for .EXE programs because they have code segments with segment addresses different from that of the PSP.

The only problem with Terminate Process with Return Code is that it is not available under MS-DOS versions earlier than 2.0, so it cannot be used if a program must be compatible with early MS-DOS versions. However, Figure 4-3 shows how a program can use the approved termination method when available but still remain pre-2.0 compatible. *See* The Warm Boot/Terminate Vector below.

```
TEXT    SEGMENT PARA PUBLIC 'CODE'

        ASSUME  CS:TEXT,DS:NOTHING,ES:NOTHING,SS:NOTHING

TERM_VECTOR     DD      ?

ENTRY_PROC      PROC    FAR

;save pointer to termination vector in PSP

        MOV     WORD PTR CS:TERM_VECTOR+0,0000h ;save offset of Warm Boot vector
        MOV     WORD PTR CS:TERM_VECTOR+2,DS    ;save segment address of PSP
```

Figure 4-3. Terminating properly under any MS-DOS version. *(more)*

```
;***** Place main task here *****

;determine which MS-DOS version is active, take jump if 2.0 or later

        MOV     AH,30h          ;load Get MS-DOS Version Number function code
        INT     21h             ;call MS-DOS to get version number
        OR      AL,AL           ;see if pre-2.0 MS-DOS
        JNZ     TERM_0200       ;jump if 2.0 or later

;terminate under pre-2.0 MS-DOS

        JMP     CS:TERM_VECTOR  ;jump to Warm Boot vector in PSP

;terminate under MS-DOS 2.0 or later

TERM_0200:
        MOV     AX,4C00h        ;load MS-DOS termination function code
                                ;and return code
        INT     21h             ;call MS-DOS to terminate

ENTRY_PROC      ENDP

TEXT    ENDS

        END     ENTRY_PROC      ;define entry point
```

Figure 4-3. Continued.

The Terminate Program interrupt

Before MS-DOS version 2.0, terminating with an approved method meant executing an INT 20H instruction, the Terminate Program interrupt. The INT 20H instruction was replaced as the approved termination method for two primary reasons: First, it did not provide a means whereby programs could return a termination code; second, CS had to point to the PSP before the INT 20H instruction was executed.

The restriction placed on the value of CS at termination did not pose a problem for .COM programs because they execute with CS pointing to the beginning of the PSP. A .EXE program, on the other hand, executes with CS pointing to various code segments of the program, and the value of CS cannot be changed arbitrarily when the program is ready to terminate. Because of this, few .EXE programs attempt simply to execute a Terminate Program interrupt from directly within their own code segments. Instead, they usually use the termination method discussed next.

The Warm Boot/Terminate vector

The earlier discussion of the structure of the PSP briefly covered one older method a .EXE program can use to terminate: Offset 00H within the PSP contains an INT 20H instruction to which the program can jump in order to terminate. MS-DOS adopted this technique to support the many CP/M programs ported to MS-DOS. Under CP/M, this PSP location was referred to as the Warm Boot vector because the CP/M operating system was always reloaded from disk (rebooted) whenever a program terminated.

Because offset 00H in the PSP contains an INT 20H instruction, jumping to that location terminates a program in the same manner as an INT 20H included directly within the program, but with one important difference: By jumping to PSP:0000H, the program sets the CS register to point to the beginning of the PSP, thereby satisfying the only restriction imposed on executing the Terminate Program interrupt. The discussion of MS-DOS Function 4CH gave an example of how a .EXE program can terminate via PSP:0000H. The example first asks MS-DOS for its version number and then terminates via PSP:0000H only under versions of MS-DOS earlier than 2.0. Programs can also use PSP:0000H under MS-DOS versions 2.0 and later; the example uses Function 4CH simply because it is preferred under the later MS-DOS versions.

The RET instruction

The other popular method used by CP/M programs to terminate involved simply executing a RET instruction. This worked because CP/M pushed the address of the Warm Boot vector onto the stack before giving the program control. MS-DOS provides this support only for .COM-style programs; it does *not* push a termination address onto the stack before giving .EXE programs control.

The programmer who wants to use the RET instruction to return to MS-DOS can use the variation of the Figure 4-3 listing shown in Figure 4-4.

```
TEXT    SEGMENT PARA PUBLIC 'CODE'

        ASSUME  CS:TEXT,DS:NOTHING,ES:NOTHING,SS:NOTHING

ENTRY_PROC      PROC    FAR     ;make proc FAR so RET will be FAR

;Push pointer to termination vector in PSP
        PUSH    DS              ;push PSP's segment address
        XOR     AX,AX           ;ax = 0 = offset of Warm Boot vector in PSP
        PUSH    AX              ;push Warm Boot vector offset

;***** Place main task here *****

;Determine which MS-DOS version is active, take jump if 2.0 or later

        MOV     AH,30h          ;load Get MS-DOS Version Number function code
        INT     21h             ;call MS-DOS to get version number
        OR      AL,AL           ;see if pre-2.0 MS-DOS
        JNZ     TERM_0200       ;jump if 2.0 or later

;Terminate under pre-2.0 MS-DOS (this is a FAR proc, so RET will be FAR)
        RET                     ;pop PSP:00H into CS:IP to terminate
```

Figure 4-4. Using RET to return control to MS-DOS.

(more)

```
;Terminate under MS-DOS 2.0 or later
TERM_0200:
        MOV     AX,4C00h        ;AH = MS-DOS Terminate Process with Return Code
                                ;function code, AL = return code of 00H
        INT     21h             ;call MS-DOS to terminate

ENTRY_PROC      ENDP

TEXT    ENDS

        END     ENTRY_PROC      ;declare the program's entry point
```

Figure 4-4. Continued.

The Terminate Process function

The final method for terminating a .EXE program is Interrupt 21H Function 00H (Terminate Process). This method maintains the same restriction as all other older termination methods: CS must point to the PSP. Because of this restriction, .EXE programs typically avoid this method in favor of terminating via PSP:0000H, as discussed above for programs executing under versions of MS-DOS earlier than 2.0.

Terminating and staying resident

A .EXE program can use any of several additional termination methods to return control to MS-DOS but still remain resident within memory to service a special event. *See* PROGRAMMING IN THE MS-DOS ENVIRONMENT: CUSTOMIZING MS-DOS: Terminate-and-Stay-Resident Utilities.

Structure of the .EXE files

So far we've examined how the .EXE program looks in memory, how MS-DOS gives the program control of the computer, and how the program should return control to MS-DOS. Next we'll investigate what the program looks like as a disk file, before MS-DOS loads it into memory. Figure 4-5 shows the general structure of a .EXE file.

The file header

Unlike .COM program files, .EXE program files contain information that permits the .EXE program and MS-DOS to use the full capabilities of the 8086 family of microprocessors. The linker places all this extra information in a header at the start of the .EXE file. Although the .EXE file structure could easily accommodate a header as small as 32 bytes, the linker never creates a header smaller than 512 bytes. (This minimum header size corresponds to the standard record size preferred by MS-DOS.) The .EXE file header contains the following information, which MS-DOS reads into a temporary work area in memory for use while loading the .EXE program:

00–01H (.EXE Signature) MS-DOS does not rely on the extension (.EXE or .COM) to determine whether a file contains a .COM or a .EXE program. Instead, MS-DOS recognizes the file as a .EXE program if the first 2 bytes in the header contain the signature 4DH 5AH

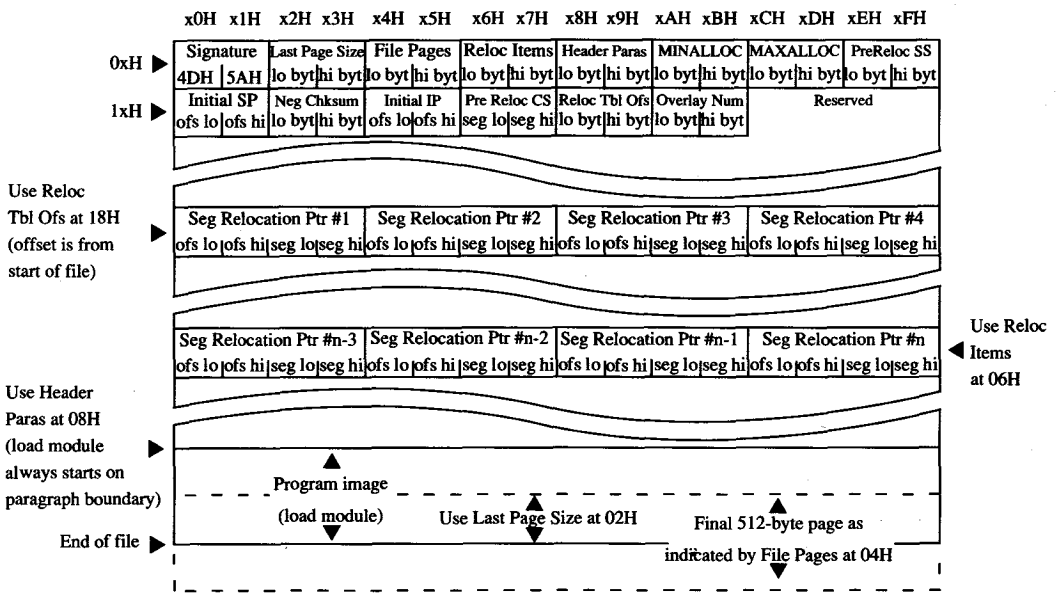

Figure 4-5. Structure of a .EXE file.

(ASCII characters *M* and *Z*). If either or both of the signature bytes contain other values, MS-DOS assumes the file contains a .COM program, regardless of the extension. The reverse is not necessarily true — that is, MS-DOS does not accept the file as a .EXE program simply because the file begins with a .EXE signature. The file must also pass several other tests.

02–03H (Last Page Size) The word at this location indicates the actual number of bytes in the final 512-byte page of the file. This word combines with the following word to determine the actual size of the file.

04–05H (File Pages) This word contains a count of the total number of 512-byte pages required to hold the file. If the file contains 1024 bytes, this word contains the value 0002H; if the file contains 1025 bytes, this word contains the value 0003H. The previous word (Last Page Size, 02–03H) is used to determine the number of valid bytes in the final 512-byte page. Thus, if the file contains 1024 bytes, the Last Page Size word contains 0000H because no bytes overflow into a final partly used page; if the file contains 1025 bytes, the Last Page Size word contains 0001H because the final page contains only a single valid byte (the 1025th byte).

06–07H (Relocation Items) This word gives the number of entries that exist in the relocation pointer table. *See* Relocation Pointer Table below.

08–09H (Header Paragraphs) This word gives the size of the .EXE file header in 16-byte paragraphs. It indicates the offset of the program's compiled/assembled and linked image (the load module) within the .EXE file. Subtracting this word from the two file-size words starting at 02H and 04H reveals the size of the program's image. The header always spans an even multiple of 16-byte paragraphs. For example, if the file consists of a 512-byte header and a 513-byte program image, then the file's total size is 1025 bytes. As discussed before, the Last Page Size word (02–03H) will contain 0001H and the File Pages word (04–05H) will contain 0003H. Because the header is 512 bytes, the Header Paragraphs word (08–09H) will contain 32 (0020H). (That is, 32 paragraphs times 16 bytes per paragraph totals 512 bytes.) By subtracting the 512 bytes of the header from the 1025-byte total file size, the size of the program's image can be determined — in this case, 513 bytes.

0A–0BH (MINALLOC) This word indicates the minimum number of 16-byte paragraphs the program requires to begin execution ***in addition to*** the memory required to hold the program's image. MINALLOC normally represents the total size of any uninitialized data and/or stack segments linked at the end of the program. LINK excludes the space reserved by these fields from the end of the .EXE file to avoid wasting disk space. If not enough memory remains to satisfy MINALLOC when loading the program, MS-DOS returns an error to the process trying to load the program. If the process is COMMAND.COM, COMMAND.COM then displays a *Program too big to fit in memory* error message. The EXEMOD utility can alter this field if desired. *See* Modifying the .EXE File Header below.

0C–0DH (MAXALLOC) This word indicates the maximum number of 16-byte paragraphs the program would like allocated to it before it begins execution. MAXALLOC indicates *additional* memory desired beyond that required to hold the program's image. MS-DOS uses this value to allocate MAXALLOC extra paragraphs, if available. If MAXALLOC paragraphs are not available, the program receives the largest memory block available — at least MINALLOC additional paragraphs. The programmer could use the MAXALLOC field to request that MS-DOS allocate space for use as a print buffer or as a program-maintained heap, for example.

Unless otherwise specified with the /CPARMAXALLOC switch at link time, the linker sets MAXALLOC to FFFFH. This causes MS-DOS to allocate all of the largest block of memory it has available to the program. To make the program compatible with multitasking supervisor programs, the programmer should use /CPARMAXALLOC to set the true maximum number of extra paragraphs the program desires. The EXEMOD utility can also be used to alter this field.

Note: If both MINALLOC and MAXALLOC have been set to 0000H, MS-DOS loads the program as high in memory as possible. LINK sets these fields to 0000H if the /HIGH switch was used; the EXEMOD utility can also be used to modify these fields.

0E–0FH (Initial SS Value) This word contains the paragraph address of the stack segment relative to the start of the load module. At load time, MS-DOS relocates this value by adding the program's start segment address to it, and the resulting value is placed in the SS register before giving the program control. (The start segment corresponds to the first segment boundary in memory following the PSP.)

10–11H (Initial SP Value) This word contains the absolute value that MS-DOS loads into the SP register before giving the program control. Because MS-DOS always loads programs starting on a segment address boundary, and because the linker knows the size of the stack segment, the linker is able to determine the correct SP offset at link time; therefore, MS-DOS does not need to adjust this value at load time. The EXEMOD utility can be used to alter this field.

12–13H (Complemented Checksum) This word contains the one's complement of the summation of all words in the .EXE file. Current versions of MS-DOS basically ignore this word when they load a .EXE program; however, future versions might not. When LINK generates a .EXE file, it adds together all the contents of the .EXE file (including the .EXE header) by treating the entire file as a long sequence of 16-bit words. During this addition, LINK gives the Complemented Checksum word (12–13H) a temporary value of 0000H. If the file consists of an odd number of bytes, then the final byte is treated as a word with a high byte of 00H. Once LINK has totaled all words in the .EXE file, it performs a one's complement operation on the total and records the answer in the .EXE file header at offsets 12–13H. The validity of a .EXE file can then be checked by performing the same word-totaling process as LINK performed. The total should be FFFFH, because the total will include LINK's calculated complemented checksum, which is designed to give the file the FFFFH total.

An example 7-byte .EXE file illustrates how .EXE file checksums are calculated. (This is a totally fictitious file, because .EXE headers are never smaller than 512 bytes.) If this fictitious file contained the bytes 8CH C8H 8EH D8H BAH 10H B4H, then the file's total would be calculated using C88CH+D88EH+10BAH+00B4H=1B288H. (Overflow past 16 bits is ignored, so the value is interpreted as B288H.) If this were a valid .EXE file, then the B288H total would have been FFFFH instead.

14–15H (Initial IP Value) This word contains the absolute value that MS-DOS loads into the IP register in order to transfer control to the program. Because MS-DOS always loads programs starting on a segment address boundary, the linker can calculate the correct IP offset from the initial CS register value at link time; therefore, MS-DOS does not need to adjust this value at load time.

16–17H (Pre-Relocated Initial CS Value) This word contains the initial value, relative to the start of the load module, that MS-DOS places in the CS register to give the .EXE program control. MS-DOS adjusts this value in the same manner as the initial SS value before loading it into the CS register.

18–19H (Relocation Table Offset) This word gives the offset from the start of the file to the relocation pointer table. This word must be used to locate the relocation pointer table, because variable-length information pertaining to program overlays can occur before the table, thus causing the position of the table to vary.

1A–1BH (Overlay Number) This word is normally set to 0000H, indicating that the .EXE file consists of the resident, or primary, part of the program. This number changes only in files containing programs that use overlays, which are sections of a program that remain

on disk until the program actually requires them. These program sections are loaded into memory by special overlay managing routines included in the run-time libraries supplied with some Microsoft high-level-language compilers.

The preceding section of the header (00–1BH) is known as the formatted area. Optional information used by high-level-language overlay managers can follow this formatted area. Unless the program in the .EXE file incorporates such information, the relocation pointer table immediately follows the formatted header area.

Relocation Pointer Table The relocation pointer table consists of a list of pointers to words within the .EXE program image that MS-DOS must adjust before giving the program control. These words consist of references made by the program to the segments that make up the program. MS-DOS must adjust these segment address references when it loads the program, because it can load the program into memory starting at any segment address boundary.

Each pointer in the table consists of a doubleword. The first word contains an offset from the segment address given in the second word, which in turn indicates a segment address relative to the start of the load module. Together, these two words point to a third word within the load module that must have the start segment address added to it. (The start segment corresponds to the segment address at which MS-DOS started loading the program's

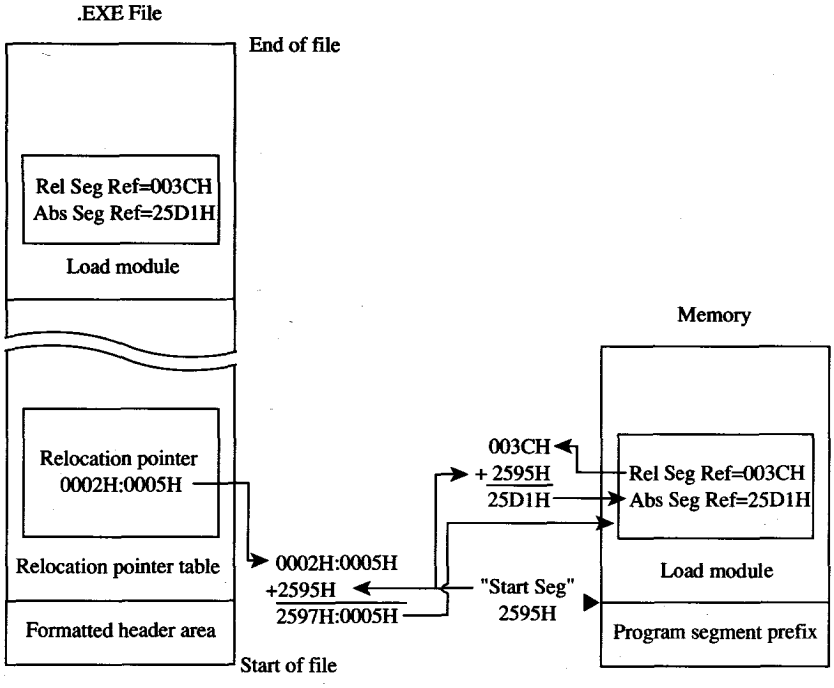

Figure 4-6. The .EXE file relocation procedure.

image, immediately following the PSP.) Figure 4-6 shows the entire procedure MS-DOS performs for each relocation table entry.

The load module

The load module starts where the .EXE header ends and consists of the fully linked image of the program. The load module appears within the .EXE file exactly as it would appear in memory if MS-DOS were to load it at segment address 0000H. The only changes MS-DOS makes to the load module involve relocating any direct segment references.

Although the .EXE file contains distinct segment images within the load module, it provides no information for separating those individual segments from one another. Existing versions of MS-DOS ignore how the program is segmented; they simply copy the load module into memory, relocate any direct segment references, and give the program control.

Loading the .EXE program

So far we've covered all the characteristics of the .EXE program as it resides in memory and on disk. We've also touched on all the steps MS-DOS performs while loading the .EXE program from disk and executing it. The following list recaps the .EXE program loading process in the order in which MS-DOS performs it:

1. MS-DOS reads the formatted area of the header (the first 1BH bytes) from the .EXE file into a work area.
2. MS-DOS determines the size of the largest available block of memory.
3. MS-DOS determines the size of the load module using the Last Page Size (offset 02H), File Pages (offset 04H), and Header Paragraphs (offset 08H) fields from the header. An example of this process is in the discussion of the Header Paragraphs field.
4. MS-DOS adds the MINALLOC field (offset 0AH) in the header to the calculated load-module size and the size of the PSP (100H bytes). If this total exceeds the size of the largest available block, MS-DOS terminates the load process and returns an error to the calling process. If the calling process was COMMAND.COM, COMMAND.COM then displays a *Program too big to fit in memory* error message.
5. MS-DOS adds the MAXALLOC field (offset 0CH) in the header to the calculated load-module size and the size of the PSP. If the memory block found earlier exceeds this calculated total, MS-DOS allocates the calculated memory size to the program from the memory block; if the calculated total exceeds the block's size, MS-DOS allocates the entire block.
6. If the MINALLOC and MAXALLOC fields both contain 0000H, MS-DOS uses the calculated load-module size to determine a start segment. MS-DOS calculates the start segment so that the load module will load into the high end of the allocated block. If either MINALLOC or MAXALLOC contains nonzero values (the normal case), MS-DOS establishes the start segment as the segment following the PSP.
7. MS-DOS loads the load module into memory starting at the start segment.

8. MS-DOS reads the relocation pointers into a work area and relocates the load module's direct segment references, as shown in Figure 4-6.
9. MS-DOS builds a PSP in the first 100H bytes of the allocated memory block. While building the two FCBs within the PSP, MS-DOS determines the initial values for the AL and AH registers.
10. MS-DOS sets the SS and SP registers to the values in the header after the start segment is added to the SS value.
11. MS-DOS sets the DS and ES registers to point to the beginning of the PSP.
12. MS-DOS transfers control to the .EXE program by setting CS and IP to the values in the header after adding the start segment to the CS value.

Controlling the .EXE program's structure

We've now covered almost every aspect of a completed .EXE program. Next, we'll discuss how to control the structure of the final .EXE program from the source level. We'll start by covering the statements provided by MASM that permit the programmer to define the structure of the program when programming in assembly language. Then we'll cover the five standard memory models provided by Microsoft's C and FORTRAN compilers (both version 4.0), which provide predefined structuring over which the programmer has limited control.

The MASM SEGMENT directive

MASM's SEGMENT directive and its associated ENDS directive mark the beginning and end of a program segment. Program segments contain collections of code or data that have offset addresses relative to the same common segment address.

In addition to the required segment name, the SEGMENT directive has three optional parameters:

segname SEGMENT [*align*] [*combine*] ['*class*']

With MASM, the contents of a segment can be defined at one point in the source file and the definition can be resumed as many times as necessary throughout the remainder of the file. When MASM encounters a SEGMENT directive with a *segname* it has previously encountered, it simply resumes the segment definition where it left off. This occurs regardless of the *combine* type specified in the SEGMENT directive—the *combine* type influences only the actions of the linker. *See* The *combine* Type Parameter below.

The *align* type parameter

The optional *align* parameter lets the programmer send the linker an instruction on how to align a segment within memory. In reality, the linker can align the segment only in relation to the start of the program's load module, but the result remains the same because MS-DOS always loads the module aligned on a paragraph (16-byte) boundary. (The PAGE *align* type creates a special exception, as discussed below.)

The following alignment types are permitted:

BYTE This *align* type instructs the linker to start the segment on the byte immediately following the previous segment. BYTE alignment prevents any wasted memory between the previous segment and the BYTE-aligned segment.

A minor disadvantage to BYTE alignment is that the 8086-family segment registers might not be able to directly address the start of the segment in all cases. Because they can address only on paragraph boundaries, the segment registers may have to point as many as 15 bytes behind the start of the segment. This means that the segment size should not be more than 15 bytes short of 64 KB. The linker adjusts offset and segment address references to compensate for differences between the physical segment start and the paragraph addressing boundary.

Another possible concern is execution speed on true 16-bit 8086-family microprocessors. When using non-8088 microprocessors, a program can actually run faster if the instructions and word data fields within segments are aligned on word boundaries. This permits the 16-bit processors to fetch full words in a single memory read, rather than having to perform two single-byte reads. The EVEN directive tells MASM to align instructions and data fields on word boundaries; however, MASM can establish this alignment only in relation to the start of the segment, so the entire segment must start aligned on a word or larger boundary to guarantee alignment of the items within the segment.

WORD This *align* type instructs the linker to start the segment on the next word boundary. Word boundaries occur every 2 bytes and consist of all even addresses (addresses in which the least significant bit contains a zero). WORD alignment permits alignment of data fields and instructions within the segment on word boundaries, as discussed for the BYTE alignment type. However, the linker may have to waste 1 byte of memory between the previous segment and the word-aligned segment in order to position the new segment on a word boundary.

Another minor disadvantage to WORD alignment is that the 8086-family segment registers might not be able to directly address the start of the segment in all cases. Because they can address only on paragraph boundaries, the segment registers may have to point as many as 14 bytes behind the start of the segment. This means that the segment size should not be more than 14 bytes short of 64 KB. The linker adjusts offset and segment address references to compensate for differences between the physical segment start and the paragraph addressing boundary.

PARA This *align* type instructs the linker to start the segment on the next paragraph boundary. The segments default to PARA if no alignment type is specified. Paragraph boundaries occur every 16 bytes and consist of all addresses with hexadecimal values ending in zero (0000H, 0010H, 0020H, and so forth). Paragraph alignment ensures that the segment begins on a segment register addressing boundary, thus making it possible to address a full 64 KB segment. Also, because paragraph addresses are even addresses, PARA alignment has the same advantages as WORD alignment. The only real disadvantage to PARA alignment is that the linker may have to waste as many as 15 bytes of memory between the previous segment and the paragraph-aligned segment.

PAGE This *align* type instructs the linker to start the segment on the next page boundary. Page boundaries occur every 256 bytes and consist of all addresses in which the low address byte equals zero (0000H, 0100H, 0200H, and so forth). PAGE alignment ensures

only that the linker positions the segment on a page boundary relative to the start of the load module. Unfortunately, this does not also ensure alignment of the segment on an absolute page within memory, because MS-DOS only guarantees alignment of the entire load module on a paragraph boundary.

When a programmer declares pieces of a segment with the same name in different source modules, the *align* type specified for each segment piece influences the alignment of that specific piece of the segment. For example, assume the following two segment declarations appear in different source modules:

```
_DATA   SEGMENT PARA PUBLIC 'DATA'
        DB      '123'
_DATA   ENDS

_DATA   SEGMENT PARA PUBLIC 'DATA'
        DB      '456'
_DATA   ENDS
```

The linker starts by aligning the first segment piece located in the first object module on a paragraph boundary, as requested. When the linker encounters the second segment piece in the second object module, it aligns that piece on the first paragraph boundary following the first segment piece. This results in a 13-byte gap between the first segment piece and the second. The segment pieces must exist in separate source modules for this to occur. If the segment pieces exist in the same source module, MASM assumes that the second segment declaration is simply a resumption of the first and creates an object module with segment declarations equivalent to the following:

```
_DATA   SEGMENT PARA PUBLIC 'DATA'
        DB      '123'
        DB      '456'
_DATA   ENDS
```

The *combine* type parameter

The optional *combine* parameter allows the programmer to send directions to the linker on how to combine segments with the same *segname* occurring in different object modules. If no *combine* type is specified, the linker treats such segments as if each had a different *segname*. The *combine* type has no effect on the relationship of segments with different *segnames*. MASM and LINK both support the following *combine* types:

PUBLIC This *combine* type instructs the linker to concatenate multiple segments having the same *segname* into a single contiguous segment. The linker adjusts any address references to labels within the concatenated segments to reflect the new position of those labels relative to the start of the combined segment. This *combine* type is useful for accessing code or data in different source modules using a common segment register value.

STACK This *combine* type operates similarly to the PUBLIC *combine* type, except for two additional effects: The STACK type tells the linker that this segment comprises part of the program's stack and initialization data contained within STACK segments is handled differently than in PUBLIC segments. Declaring segments with the STACK *combine* type permits the linker to determine the initial SS and SP register values it places in the .EXE

file header. Normally, a programmer would declare only one STACK segment in one of the source modules. If pieces of the stack are declared in different source modules, the linker will concatenate them in the same fashion as PUBLIC segments. However, initialization data declared within any STACK segment is placed at the high end of the combined STACK segments on a module-by-module basis. Thus, each successive module's initialization data overlays the previous module's data. At least one segment must be declared with the STACK *combine* type; otherwise, the linker will issue a warning message because it cannot determine the program's initial SS and SP values. (The warning can be ignored if the program itself initializes SS and SP.)

COMMON This *combine* type instructs the linker to overlap multiple segments having the same *segname*. The length of the resulting segment reflects the length of the longest segment declared. If any code or data is declared in the overlapping segments, the data contained in the final segments linked replaces any data in previously loaded segments. This *combine* type is useful when a data area is to be shared by code in different source modules.

MEMORY Microsoft's LINK treats this *combine* type the same as it treats the PUBLIC type. MASM, however, supports the MEMORY type for compatibility with other linkers that use Intel's definition of a MEMORY *combine* type.

AT address This *combine* type instructs LINK to pretend that the segment will reside at the absolute segment *address*. LINK then adjusts all address references to the segment in accordance with the masquerade. LINK will *not* create an image of the segment in the load module, and it will ignore any data defined within the segment. This behavior is consistent with the fact that MS-DOS does not support the loading of program segments into absolute memory segments. All programs must be able to execute from any segment address at which MS-DOS can find available memory. The SEGMENT AT address *combine* type is useful for creating templates of various areas in memory outside the program. For instance, *SEGMENT AT 0000H* could be used to create a template of the 8086-family interrupt vectors. Because data contained within SEGMENT AT address segments is suppressed by LINK and not by MASM (which places the data in the object module), it is possible to use .OBJ files generated by MASM with another linker that supports ROM or other absolute code generation should the programmer require this specialized capability.

The *class* type parameter

The *class* parameter provides the means to organize different segments into classifications. For instance, here are three source modules, each with its own separate code and data segments:

```
;Module "A"
A_DATA  SEGMENT PARA PUBLIC 'DATA'
;Module "A" data fields
A_DATA  ENDS
A_CODE  SEGMENT PARA PUBLIC 'CODE'
;Module "A" code
A_CODE  ENDS
        END
```

(more)

```
;Module "B"
B_DATA  SEGMENT PARA PUBLIC 'DATA'
;Module "B" data fields
B_DATA  ENDS
B_CODE  SEGMENT PARA PUBLIC 'CODE'
;Module "B" code
B_CODE  ENDS
        END

;Module "C"
C_DATA  SEGMENT PARA PUBLIC 'DATA'
;Module "C" data fields
C_DATA  ENDS
C_CODE  SEGMENT PARA PUBLIC 'CODE'
;Module "C" code
C_CODE  ENDS
        END
```

If the 'CODE' and 'DATA' *class* types are removed from the SEGMENT directives shown above, the linker organizes the segments as it encounters them. If the programmer specifies the modules to the linker in alphabetic order, the linker produces the following segment ordering:

```
A_DATA
A_CODE
B_DATA
B_CODE
C_DATA
C_CODE
```

However, if the programmer specifies the *class* types shown in the sample source modules, the linker organizes the segments by classification as follows:

```
'DATA' class:    A_DATA
                 B_DATA
                 C_DATA

'CODE' class:    A_CODE
                 B_CODE
                 C_CODE
```

Notice that the linker still organizes the classifications in the order in which it encounters the segments belonging to the various classifications. To completely control the order in which the linker organizes the segments, the programmer must use one of three basic approaches. The preferred method involves using the /DOSSEG switch with the linker. This produces the segment ordering shown in Figure 4-1. The second method involves creating a special source module that contains empty SEGMENT–ENDS blocks for all the segments declared in the various other source modules. The programmer creates the list in the order the segments are to be arranged in memory and then specifies the .OBJ file for this module as the first file for the linker to process. This procedure establishes the order of all the segments before LINK begins processing the other program modules, so the

programmer can declare segments in these other modules in any convenient order. For instance, the following source module rearranges the result of the previous example so that the linker places the 'CODE' class before the 'DATA' class:

```
A_CODE  SEGMENT PARA PUBLIC 'CODE'
A_CODE  ENDS
B_CODE  SEGMENT PARA PUBLIC 'CODE'
B_CODE  ENDS
C_CODE  SEGMENT PARA PUBLIC 'CODE'
C_CODE  ENDS

A_DATA  SEGMENT PARA PUBLIC 'DATA'
A_DATA  ENDS
B_DATA  SEGMENT PARA PUBLIC 'DATA'
B_DATA  ENDS
C_DATA  SEGMENT PARA PUBLIC 'DATA'
C_DATA  ENDS

        END
```

Rather than creating a new module, the third method places the same segment ordering list shown above at the start of the first module containing actual code or data that the programmer will be specifying for the linker. This duplicates the approach used by Microsoft's newer compilers, such as C version 4.0.

The ordering of segments within the load module has no direct effect on the linker's adjustment of address references to locations within the various segments. Only the GROUP directive and the SEGMENT directive's *combine* parameter affect address adjustments performed by the linker. *See* The MASM GROUP Directive below.

Note: Certain older versions of the IBM Macro Assembler wrote segments to the object file in alphabetic order regardless of their order in the source file. These older versions can limit efforts to control segment ordering. Upgrading to a new version of the assembler is the best solution to this problem.

Ordering segments to shrink the .EXE file

Correct segment ordering can significantly decrease the size of a .EXE program as it resides on disk. This size-reduction ordering is achieved by placing all uninitialized data fields in their own segments and then controlling the linker's ordering of the program's segments so that the uninitialized data field segments all reside at the end of the program. When the program modules are assembled, MASM places information in the object modules to tell the linker about initialized and uninitialized areas of all segments. The linker then uses this information to prevent the writing of uninitialized data areas that occur at the end of the program image as part of the resulting .EXE file. To account for the memory space required by these fields, the linker also sets the MINALLOC field in the .EXE file header to represent the data area not written to the file. MS-DOS then uses the MINALLOC field to reallocate this missing space when loading the program.

The MASM GROUP directive

The MASM GROUP directive can also have a strong impact on a .EXE program. However, the GROUP directive has *no* effect on the arrangement of program segments within memory. Rather, GROUP associates program segments for addressing purposes.

The GROUP directive has the following syntax:

grpname GROUP *segname,segname,segname,...*

This directive causes the linker to adjust all address references to labels within any specified *segname* to be relative to the start of the declared group. The start of the group is determined at link time. The group starts with whichever of the segments in the GROUP list the linker places lowest in memory.

That the GROUP directive neither causes nor requires contiguous arrangement of the grouped segments creates some interesting, although not necessarily desirable, possibilities. For instance, it permits the programmer to locate segments not belonging to the declared group between segments that do belong to the group. The only restriction imposed on the declared group is that the last byte of the last segment in the group must occur within 64 KB of the start of the group. Figure 4-7 illustrates this type of segment arrangement:

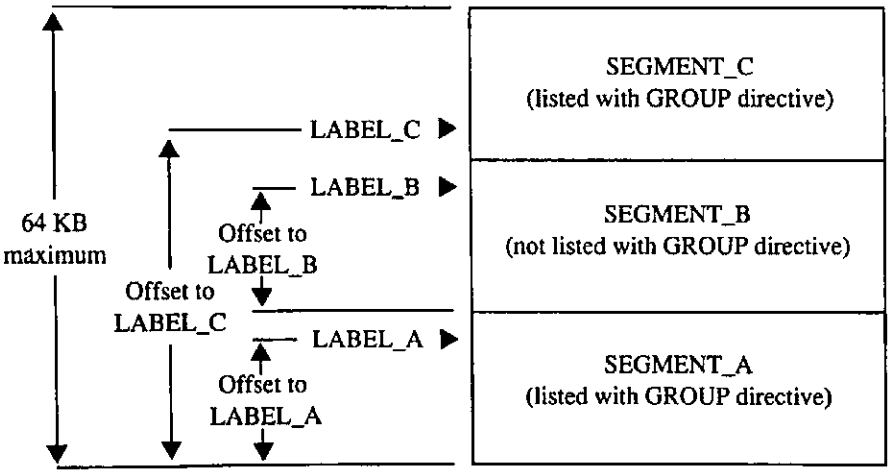

Figure 4-7. Noncontiguous segments in the same GROUP.

Warning: One of the most confusing aspects of the GROUP directive relates to MASM's OFFSET operator. The GROUP directive affects only the offset addresses generated by such direct addressing instructions as

```
MOV     AX,FIELD_LABEL
```

but it has no effect on immediate address values generated by such instructions as

```
MOV     AX,OFFSET FIELD_LABEL
```

Using the OFFSET operator on labels contained within grouped segments requires the following approach:

```
MOV     AX,OFFSET GROUP_NAME:FIELD_LABEL
```

The programmer must *explicitly* request the offset from the group base, because MASM defines the result of the OFFSET operator to be the offset of the label from the start of its segment, not its group.

Structuring a small program with SEGMENT and GROUP

Now that we have analyzed the functions performed by the SEGMENT and GROUP directives, we'll put both directives to work structuring a skeleton program. The program, shown in Figures 4-8, 4-9, and 4-10, consists of three source modules (MODULE_A, MODULE_B, and MODULE_C), each using the following four program segments:

Segment	Definition
_TEXT	The code or program text segment
_DATA	The standard data segment containing preinitialized data fields the program might change
CONST	The constant data segment containing constant data fields the program will not change
_BSS	The "block storage segment/space" segment containing uninitialized data fields*

*Programmers familiar with the IBM 1620/1630 or CDC 6000 and Cyber assemblers may recognize BSS as "block started at symbol," which reflects an equally appropriate, although somewhat more elaborate, definition of the abbreviation. Other common translations of BSS, such as "blank static storage," misrepresent the segment name, because blanking of BSS segments does not occur—the memory contains undetermined values when the program begins execution.

```
;Source Module MODULE_A

;Predeclare all segments to force the linker's segment ordering **************

_TEXT   SEGMENT BYTE PUBLIC 'CODE'
_TEXT   ENDS

_DATA   SEGMENT WORD PUBLIC 'DATA'
_DATA   ENDS

CONST   SEGMENT WORD PUBLIC 'CONST'
CONST   ENDS

_BSS    SEGMENT WORD PUBLIC 'BSS'
_BSS    ENDS
```

Figure 4-8. Structuring a .EXE program: MODULE_A.

(more)

```
STACK   SEGMENT PARA STACK 'STACK'
STACK   ENDS

DGROUP  GROUP   _DATA,CONST,_BSS,STACK

;Constant declarations ***************************************************

CONST   SEGMENT WORD PUBLIC 'CONST'

CONST_FIELD_A   DB      'Constant A'    ;declare a MODULE_A constant

CONST   ENDS

;Preinitialized data fields ***********************************************

_DATA   SEGMENT WORD PUBLIC 'DATA'

DATA_FIELD_A    DB      'Data A'        ;declare a MODULE_A preinitialized field

_DATA   ENDS

;Uninitialized data fields *************************************************

_BSS    SEGMENT WORD PUBLIC 'BSS'

BSS_FIELD_A     DB      5 DUP(?)        ;declare a MODULE_A uninitialized field

_BSS    ENDS

;Program text *************************************************************

_TEXT   SEGMENT BYTE PUBLIC 'CODE'

        ASSUME  CS:_TEXT,DS:DGROUP,ES:NOTHING,SS:NOTHING

        EXTRN   PROC_B:NEAR             ;label is in _TEXT segment (NEAR)
        EXTRN   PROC_C:NEAR             ;label is in _TEXT segment (NEAR)

PROC_A  PROC    NEAR

        CALL    PROC_B                  ;call into MODULE_B
        CALL    PROC_C                  ;call into MODULE_C

        MOV     AX,4C00H                ;terminate (MS-DOS 2.0 or later only)
        INT     21H

PROC_A  ENDP

_TEXT   ENDS
```

Figure 4-8. Continued. *(more)*

```
;Stack ************************************************************************

STACK   SEGMENT PARA STACK 'STACK'

        DW      128 DUP(?)              ;declare some space to use as stack
STACK_BASE      LABEL   WORD

STACK   ENDS

        END     PROC_A                  ;declare PROC_A as entry point
```

Figure 4-8. Continued.

```
;Source Module MODULE_B

;Constant declarations ********************************************************

CONST   SEGMENT WORD PUBLIC 'CONST'

CONST_FIELD_B   DB      'Constant B'    ;declare a MODULE_B constant

CONST   ENDS

;Preinitialized data fields ***************************************************

_DATA   SEGMENT WORD PUBLIC 'DATA'

DATA_FIELD_B    DB      'Data B'        ;declare a MODULE_B preinitialized field

_DATA   ENDS

;Uninitialized data fields ****************************************************

_BSS    SEGMENT WORD PUBLIC 'BSS'

BSS_FIELD_B     DB      5 DUP(?)        ;declare a MODULE_B uninitialized field

_BSS    ENDS

;Program text *****************************************************************

DGROUP  GROUP   _DATA,CONST,_BSS

_TEXT   SEGMENT BYTE PUBLIC 'CODE'

        ASSUME  CS:_TEXT,DS:DGROUP,ES:NOTHING,SS:NOTHING
```

Figure 4-9. Structuring a .EXE program: MODULE_B. *(more)*

```
        PUBLIC  PROC_B                  ;reference in MODULE_A
PROC_B  PROC    NEAR

        RET

PROC_B  ENDP

_TEXT   ENDS

        END
```

Figure 4-9. Continued.

```
;Source Module MODULE_C

;Constant declarations ***************************************************

CONST   SEGMENT WORD PUBLIC 'CONST'

CONST_FIELD_C   DB      'Constant C'    ;declare a MODULE_C constant

CONST   ENDS

;Preinitialized data fields ***********************************************

_DATA   SEGMENT WORD PUBLIC 'DATA'

DATA_FIELD_C    DB      'Data C'        ;declare a MODULE_C preinitialized field

_DATA   ENDS

;Uninitialized data fields ************************************************

_BSS    SEGMENT WORD PUBLIC 'BSS'

BSS_FIELD_C     DB      5 DUP(?)        ;declare a MODULE_C uninitialized field

_BSS    ENDS

;Program text *************************************************************

DGROUP  GROUP   _DATA,CONST,_BSS

_TEXT   SEGMENT BYTE PUBLIC 'CODE'

        ASSUME  CS:_TEXT,DS:DGROUP,ES:NOTHING,SS:NOTHING
```

Figure 4-10. Structuring a .EXE program: MODULE_C.

(more)

```
        PUBLIC  PROC_C                  ;referenced in MODULE_A
PROC_C  PROC    NEAR

        RET

PROC_C  ENDP

_TEXT   ENDS

        END
```

Figure 4-10. Continued.

This example creates a small memory model program image, so the linked program can have only a single code segment and a single data segment — the simplest standard form of a .EXE program. *See* Using Microsoft's Contemporary Memory Models below.

In addition to declaring the four segments already discussed, MODULE_A declares a STACK segment in which to define a block of memory for use as the program's stack and also defines the linking order of the five segments. Defining the linking order leaves the programmer free to declare the segments in any order when defining the segment contents — a necessity because the assembler has difficulty assembling programs that use forward references.

With Microsoft's MASM and LINK on the same disk with the .ASM files, the following commands can be made into a batch file:

```
MASM STRUCA;
MASM STRUCB;
MASM STRUCC;
LINK STRUCA+STRUCB+STRUCC/M;
```

These commands will assemble and link all the .ASM files listed, producing the memory map report file STRUCA.MAP shown in Figure 4-11.

```
 Start  Stop   Length Name                   Class
 00000H 0000CH 0000DH _TEXT                 CODE
 0000EH 0001FH 00012H _DATA                 DATA
 00020H 0003DH 0001EH CONST                 CONST
 0003EH 0004EH 00011H _BSS                  BSS
 00050H 0014FH 00100H STACK                 STACK

 Origin   Group
 0000:0   DGROUP

  Address         Publics by Name

 0000:000B       PROC_B
 0000:000C       PROC_C
```

Figure 4-11. Structuring a .EXE program: memory map report. *(more)*

```
  Address          Publics by Value

 0000:000B        PROC_B
 0000:000C        PROC_C
Program entry point at 0000:0000
```

Figure 4-11. Continued.

The above memory map report represents the memory diagram shown in Figure 4-12.

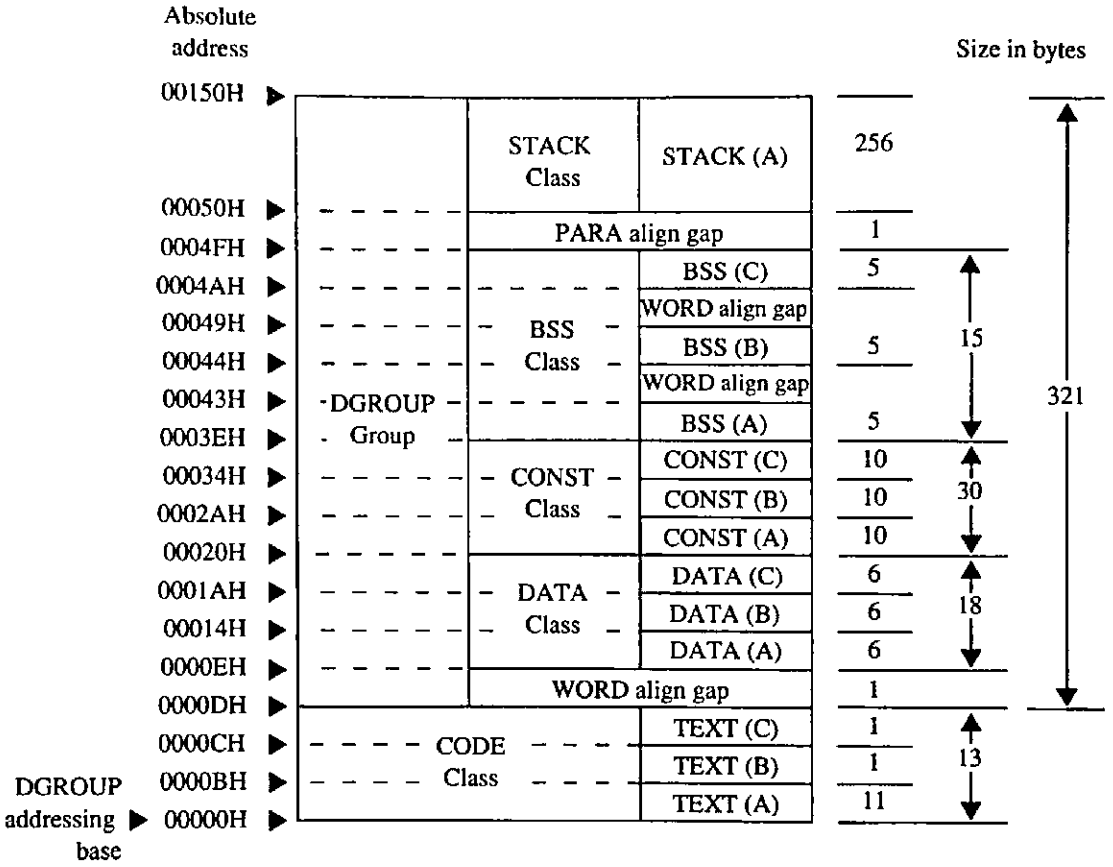

Figure 4-12. Structure of the sample .EXE program.

Using Microsoft's contemporary memory models

Now that we've analyzed the various aspects of designing assembly-language .EXE programs, we can look at how Microsoft's high-level-language compilers create .EXE programs from high-level-language source files. Even assembly-language programmers will find this discussion of interest and should seriously consider using the five standard memory models outlined here.

This discussion is based on the Microsoft C Compiler version 4.0, which, along with the Microsoft FORTRAN Compiler version 4.0, incorporates the most contemporary code generator currently available. These newer compilers generate code based on three to five

of the following standard programmer-selectable program structures, referred to as memory models. The discussion of each of these memory models will center on the model's use with the Microsoft C Compiler and will close with comments regarding any differences for the Microsoft FORTRAN Compiler.

Small (C compiler switch /AS) This model, the default, includes only a single code segment and a single data segment. All code must fit within 64 KB, and all data must fit within an additional 64 KB. Most C program designs fall into this category. Data can exceed the 64 KB limit only if the far and huge attributes are used, forcing the compiler to use far addressing, and the linker to place far and huge data items into separate segments. The data-size-threshold switch described for the compact model is ignored by the Microsoft C Compiler when used with a small model. The C compiler uses the default segment name _TEXT for all code and the default segment name _DATA for all non-far/huge data. Microsoft FORTRAN programs can generate a semblance of this model only by using the /NM (name module) and /AM (medium model) compiler switches in combination with the near attribute on all subprogram declarations.

Medium (C and FORTRAN compiler switch /AM) This model includes only a single data segment but breaks the code into multiple code segments. All data must fit within 64 KB, but the 64 KB restriction on code size applies only on a module-by-module basis. Data can exceed the 64 KB limit only if the far and huge attributes are used, forcing the compiler to use far addressing, and the linker to place far and huge data items into separate segments. The data-size-threshold switch described for the compact model is ignored by the Microsoft C Compiler when used with a medium model. The compiler uses the default segment name _DATA for all non-far/huge data and the template *module*_TEXT to create names for all code segments. The *module* element of *module*_TEXT indicates where the compiler is to substitute the name of the source module. For example, if the source module HELPFUNC.C is compiled using the medium model, the compiler creates the code segment HELPFUNC_TEXT. The Microsoft FORTRAN Compiler version 4.0 directly supports the medium model.

Compact (C compiler switch /AC) This model includes only a single code segment but breaks the data into multiple data segments. All code must fit within 64 KB, but the data is allowed to consume all the remaining available memory. The Microsoft C Compiler's optional data-size-threshold switch (/Gt) controls the placement of the larger data items into additional data segments, leaving the smaller items in the default segment for faster access. Individual data items within the program cannot exceed 64 KB under the compact model without being explicitly declared huge. The compiler uses the default segment name _TEXT for all code segments and the template *module*#_DATA to create names for all data segments. The *module* element indicates where the compiler is to substitute the source module's name; the # element represents a digit that the compiler changes for each additional data segment required to hold the module's data. The compiler starts with the digit 5 and counts up. For example, if the name of the source module is HELPFUNC.C, the compiler names the first data segment HELPFUNC5_DATA. FORTRAN programs can generate a semblance of this model only by using the /NM (name module) and /AL (large model) compiler switches in combination with the near attribute on all subprogram declarations.

Large (C and FORTRAN compiler switch /AL) This model creates multiple code and data segments. The compiler treats data in the same manner as it does for the compact model and treats code in the same manner as it does for the medium model. The Microsoft FORTRAN Compiler version 4.0 directly supports the large model.

Huge (C and FORTRAN compiler switch /AH) Allocation of segments under the huge model follows the same rules as for the large model. The difference is that individual data items can exceed 64 KB. Under the huge model, the compiler generates the necessary code to index arrays or adjust pointers across segment boundaries, effectively transforming the microprocessor's segment-addressed memory into linear-addressed memory. This makes the huge model especially useful for porting a program originally written for a processor that used linear addressing. The speed penalties the program pays in exchange for this addressing freedom require serious consideration. If the program actually contains any data structures exceeding 64 KB, it probably contains only a few. In that case, it is best to avoid using the huge model by explicitly declaring those few data items as huge using the huge keyword within the source module. This prevents penalizing all the non-huge items with extra addressing math. The Microsoft FORTRAN Compiler version 4.0 directly supports the huge model.

Figure 4-13 shows an example of the segment arrangement created by a large/huge model program. The example assumes two source modules: MSCA.C and MSCB.C. Each source module specifies enough data to cause the compiler to create two extra data segments for that module. The diagram does not show all the various segments that occur as a result of linking with the run-time library or as a result of compiling with the intention of using the CodeView debugger.

Groups	Classes	Segments	
DGROUP	STACK	STACK	◀ SMCLH: Program stack
	BSS	c_common	◀ SM: All uninitialized global items, CLH: Empty
		_BSS	◀ SMCLH: All uninitialized non-far/huge items
	CONST	CONST	◀ SMCLH: Constants (floating point constraints, segment addresses, etc.)
	DATA	_DATA	◀ SMCLH: All items that don't end up anywhere else
	FAR_BSS	FAR_BSS	◀ SM: Nonexistent, CLH: All uninitialized global items
	FAR_DATA	MSCB6_DATA	◀ From MSCB only: SM: Far/huge items, CLH: Items larger than threshold
		MSCB5_DATA	◀ From MSCB only: SM: Far/huge items, CLH: Items larger than threshold
		MSCA6_DATA	◀ From MSCA only: SM: Far/huge items, CLH: Items larger than threshold
		MSCA5_DATA	◀ From MSCA only: SM: Far/huge items, CLH: Items larger than threshold
	CODE	TEXT	◀ SC: All code, MLH: Run-time library code only
		MSCB_TEXT	◀ SC: Nonexistent, MLH: MSCB.C Code
		MSCA_TEXT	◀ SC: Nonexistent, MLH: MSCA.C Code

S = Small model L = Large model
M = Medium model H = Huge model
C = Compact model

Figure 4-13. General structure of a Microsoft C program.

Note that if the program declares an extremely large number of small data items, it can exceed the 64 KB size limit on the default data segment (_DATA) regardless of the memory model specified. This occurs because the data items all fall below the data-size-threshold limit (compiler /Gt switch), causing the compiler to place them in the _DATA segment. Lowering the data size threshold or explicitly using the far attribute within the source modules eliminates this problem.

Modifying the .EXE file header

With most of its language compilers, Microsoft supplies a utility program called EXEMOD. *See* PROGRAMMING UTILITIES: EXEMOD. This utility allows the programmer to display and modify certain fields contained within the .EXE file header. Following are the header fields EXEMOD can modify (based on EXEMOD version 4.0):

MAXALLOC This field can be modified by using EXEMOD's /MAX switch. Because EXEMOD operates on .EXE files that have already been linked, the /MAX switch can be used to modify the MAXALLOC field in existing .EXE programs that contain the default MAXALLOC value of FFFFH, provided the programs do not rely on MS-DOS's allocating all free memory to them. EXEMOD's /MAX switch functions in an identical manner to LINK's /CPARMAXALLOC switch.

MINALLOC This field can be modified by using EXEMOD's /MIN switch. Unlike the case with the MAXALLOC field, most programs do not have an arbitrary value for MINALLOC. MINALLOC normally represents uninitialized memory and stack space the linker has compressed out of the .EXE file, so a programmer should never *reduce* the MINALLOC value within a .EXE program written by someone else. If a program requires some minimum amount of extra dynamic memory in addition to any static fields, MINALLOC can be increased to ensure that the program will have this extra memory before receiving control. If this is done, the program will not have to verify that MS-DOS allocated enough memory to meet program needs. Of course, the same result can be achieved without EXEMOD by declaring this minimum extra memory as an uninitialized field at the end of the program.

Initial SP Value This field can be modified by using the /STACK switch to increase or decrease the size of a program's stack. However, modifying the initial SP value for programs developed using Microsoft language compiler versions earlier than the following may cause the programs to fail: C version 3.0, Pascal version 3.3, and FORTRAN version 3.3. Other language compilers may have the same restriction. The /STACK switch can also be used with programs developed using MASM, provided the stack space is linked at the end of the program, but it would probably be wise to change the size of the STACK segment declaration within the program instead. The linker also provides a /STACK switch that performs the same purpose.

Note: With the /H switch set, EXEMOD displays the current values of the fields within the .EXE header. This switch should not be used with the other switches. EXEMOD also displays field values if no switches are used.

Warning: EXEMOD also functions correctly when used with packed .EXE files created using EXEPACK or the /EXEPACK linker switch. However, it is important to use the EXEMOD version shipped with the linker or EXEPACK utility. Possible future changes in the packing method may result in incompatibilities between EXEMOD and nonassociated linker/EXEPACK versions.

Patching the .EXE program using DEBUG

Every experienced programmer knows that programs always seem to have at least one unspotted error. If a program has been distributed to other users, the programmer will probably need to provide those users with corrections when such bugs come to light. One inexpensive updating approach used by many large companies consists of mailing out single-page instructions explaining how the user can patch the program to correct the problem.

Program patching usually involves loading the program file into the DEBUG utility supplied with MS-DOS, storing new bytes into the program image, and then saving the program file back to disk. Unfortunately, DEBUG cannot load a .EXE program into memory and then save it back to disk in .EXE format. The programmer must trick DEBUG into patching .EXE program files, using the procedure outlined below. *See* PROGRAMMING UTILITIES: DEBUG.

Note: Users should be reminded to make backup copies of their program before attempting the patching procedure.

1. Rename the .EXE file using a filename extension that does not have special meaning for DEBUG. (Avoid .EXE, .COM, and .HEX.) For instance, MYPROG.BIN serves well as a temporary new name for MYPROG.EXE because DEBUG does not recognize a file with a .BIN extension as anything special. DEBUG will load the entire image of MYPROG.BIN, including the .EXE header and relocation table, into memory starting at offset 100H within a .COM-style program segment (as discussed previously).
2. Locate the area within the load module section of the .EXE file image that requires patching. The previous discussion of the .EXE file image, together with compiler/assembler listings and linker memory map reports, provides the information necessary to locate the error within the .EXE file image. DEBUG loads the file image starting at offset 100H within a .COM-style program segment, so the programmer must compensate for this offset when calculating addresses within the file image. Also, the compiler listings and linker memory map reports provide addresses relative to the start of the program image within the .EXE file, not relative to the start of the file itself. Therefore, the programmer must first check the information contained in the .EXE file header to determine where the load module (the program's image) starts within the file.
3. Use DEBUG's E (Enter Data) or A (Assemble Machine Instructions) command to insert the corrections. (Normally, patch instructions to users would simply give an address at which the user should apply the patch. The user need not know how to determine the address.)
4. After the patch has been applied, simply issue the DEBUG W (Write File or Sectors) command to write the corrected image back to disk under the same filename, provided the patch has not increased the size of the program. If program size has

increased, first change the appropriate size fields in the .EXE header at the start of the file and use the DEBUG R (Display or Modify Registers) command to modify the BX and CX registers so that they contain the file image's new size. Then use the W command to write the image back to disk under the same name.

5. Use the DEBUG Q (Quit) command to return to MS-DOS command level, and then rename the file to the original .EXE filename extension.

.EXE summary

To summarize, the .EXE program and file structures provide considerable flexibility in the design of programs, providing the programmer with the necessary freedom to produce large-scale applications. Programs written using Microsoft's high-level-language compilers have access to five standardized program structure models (small, medium, compact, large, and huge). These standardized models are excellent examples of ways to structure assembly-language programs.

The .COM Program

The majority of differences between .COM and .EXE programs exist because .COM program files are not prefaced by header information. Therefore, .COM programs do not benefit from the features the .EXE header provides.

The absence of a header leaves MS-DOS with no way of knowing how much memory the .COM program requires in addition to the size of the program's image. Therefore, MS-DOS must always allocate the largest free block of memory to the .COM program, regardless of the program's true memory requirements. As was discussed for .EXE programs, this allocation of the largest block of free memory usually results in MS-DOS's allocating all remaining free memory—an action that can cause problems for multitasking supervisor programs.

The .EXE program header also includes the direct segment address relocation pointer table. Because they lack this table, .COM programs cannot make address references to the labels specified in SEGMENT directives, with the exception of SEGMENT AT address directives. If a .COM program did make these references, MS-DOS would have no way of adjusting the addresses to correspond to the actual segment address into which MS-DOS loaded the program. *See* Creating the .COM Program below.

The .COM program structure exists primarily to support the vast number of CP/M programs ported to MS-DOS. Currently, .COM programs are most often used to avoid adding the 512 bytes or more of .EXE header information onto small, simple programs that often do not exceed 512 bytes by themselves.

The .COM program structure has another advantage: Its memory organization places the PSP within the same address segment as the rest of the program. Thus, it is easier to access fields within the PSP in .COM programs.

Giving control to the .COM program

After allocating the largest block of free memory to the .COM program, MS-DOS builds a PSP in the lowest 100H bytes of the block. No difference exists between the PSP MS-DOS builds for .COM programs and the PSP it builds for .EXE programs. Also with .EXE programs, MS-DOS determines the initial values for the AL and AH registers at this time and then loads the entire .COM-file image into memory immediately following the PSP. Because .COM files have no file-size header fields, MS-DOS relies on the size recorded in the disk directory to determine the size of the program image. It loads the program exactly as it appears in the file, without checking the file's contents.

MS-DOS then sets the DS, ES, and SS segment registers to point to the start of the PSP. If able to allocate at least 64 KB to the program, MS-DOS sets the SP register to offset FFFFH + 1 (0000H) to establish an initial stack; if less than 64 KB are available for allocation to the program, MS-DOS sets the SP to 1 byte past the highest offset owned by the program. In either case, MS-DOS then pushes a single word of 0000H onto the program's stack for use in terminating the program.

Finally, MS-DOS transfers control to the program by setting the CS register to the PSP's segment address and the IP register to 0100H. This means that the program's entry point must exist at the very start of the program's image, as shown in later examples.

Figure 4-14 shows the overall structure of a .COM program as it receives control from MS-DOS.

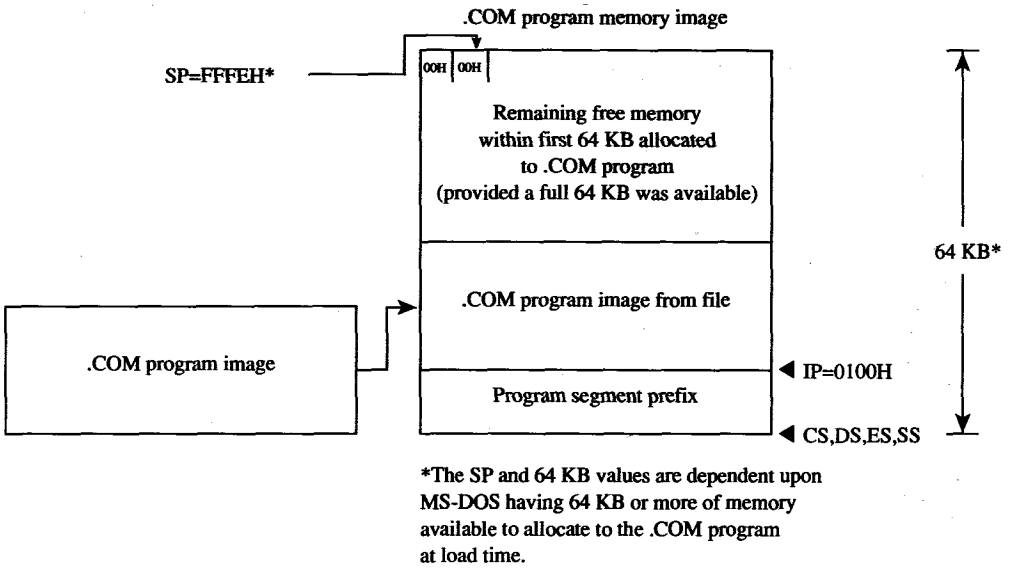

Figure 4-14. The .COM program: memory map diagram with register pointers.

Terminating the .COM program

A .COM program can use all the termination methods described for .EXE programs but should still use the MS-DOS Interrupt 21H Terminate Process with Return Code function (4CH) as the preferred method. If the .COM program must remain compatible with versions of MS-DOS earlier than 2.0, it can easily use any of the older termination methods, including those described as difficult to use from .EXE programs, because .COM programs execute with the CS register pointing to the PSP as required by these methods.

Creating the .COM program

A .COM program is created in the same manner as a .EXE program and then converted using the MS-DOS EXE2BIN utility. *See* PROGRAMMING UTILITIES: EXE2BIN.

Certain restrictions do apply to .COM programs, however. First, .COM programs cannot exceed 64 KB minus 100H bytes for the PSP minus 2 bytes for the zero word initially pushed on the stack.

Next, only a single segment — or at least a single addressing group — should exist within the program. The following two examples show ways to structure a .COM program to satisfy both this restriction and MASM's need to have data fields precede program code in the source file.

COMPROG1.ASM (Figure 4-15) declares only a single segment (*COMSEG*), so no special considerations apply when using the MASM OFFSET operator. *See* The MASM GROUP Directive above. COMPROG2.ASM (Figure 4-16) declares separate code (*CSEG*) and data (*DSEG*) segments, which the GROUP directive ties into a common addressing block. Thus, the programmer can declare data fields at the start of the source file and have the linker place the data fields segment (*DSEG*) after the code segment (*CSEG*) when it links the program, as discussed for the .EXE program structure. This second example simulates the program structuring provided under CP/M by Microsoft's old Macro-80 (M80) macro assembler and Link-80 (L80) linker. The design also expands easily to accommodate COMMON or other additional segments.

```
COMSEG  SEGMENT BYTE PUBLIC 'CODE'
        ASSUME  CS:COMSEG,DS:COMSEG,ES:COMSEG,SS:COMSEG
        ORG     0100H

BEGIN:
        JMP     START           ;skip over data fields
;Place your data fields here.

START:
;Place your program text here.
        MOV     AX,4C00H        ;terminate (MS-DOS 2.0 or later only)
        INT     21H
COMSEG  ENDS
        END     BEGIN
```

Figure 4-15. .COM program with data at start.

```
CSEG    SEGMENT BYTE PUBLIC 'CODE'      ;establish segment order
CSEG    ENDS
DSEG    SEGMENT BYTE PUBLIC 'DATA'
DSEG    ENDS
COMGRP  GROUP   CSEG,DSEG               ;establish joint address base
DSEG    SEGMENT
;Place your data fields here.
DSEG    ENDS
CSEG    SEGMENT

        ASSUME  CS:COMGRP,DS:COMGRP,ES:COMGRP,SS:COMGRP
        ORG     0100H

BEGIN:
;Place your program text here.  Remember to use
;OFFSET COMGRP:LABEL whenever you use OFFSET.
        MOV     AX,4C00H                ;terminate (MS-DOS 2.0 or later only)
        INT     21H
CSEG    ENDS
        END     BEGIN
```

Figure 4-16. .COM program with data at end.

These examples demonstrate other significant requirements for producing a functioning .COM program. For instance, the *ORG 0100H* statement in both examples tells MASM to start assembling the code at offset 100H within the encompassing segment. This corresponds to MS-DOS's transferring control to the program at IP = 0100H. In addition, the entry-point label (BEGIN) immediately follows the ORG statement and appears again as a parameter to the END statement. Together, these factors satisfy the requirement that .COM programs declare their entry point at offset 100H. If any factor is missing, the MS-DOS EXE2BIN utility will not properly convert the .EXE file produced by the linker into a .COM file. Specifically, if a .COM program declares an entry point (as a parameter to the END statement) that is at neither offset 0100H nor offset 0000H, EXE2BIN rejects the .EXE file when the programmer attempts to convert it. If the program fails to declare an entry point or declares an entry point at offset 0000H, EXE2BIN assumes that the .EXE file is to be converted to a binary image rather than to a .COM image. When EXE2BIN converts a .EXE file to a non-.COM binary file, it does not strip the extra 100H bytes the linker places in front of the code as a result of the *ORG 0100H* instruction. Thus, the program actually begins at offset 200H when MS-DOS loads it into memory, but all the program's address references will have been assembled and linked based on the 100H offset. As a result, the program—and probably the rest of the system as well—is likely to crash.

A .COM program also must not contain direct segment address references to any segments that make up the program. Thus, the .COM program cannot reference any segment labels or reference any labels as long (FAR) pointers. (This rule does not prevent the program from referencing segment labels declared using the SEGMENT AT address directive.) Following are various examples of direct segment address references that are *not* permitted as part of .COM programs:

```
PROC_A  PROC    FAR
PROC_A  ENDP.
        CALL    PROC_A          ;intersegment call
        JMP     PROC_A          ;intersegment jump
```

or

```
        EXTRN   PROC_A:FAR
        CALL    PROC_A          ;intersegment call
        JMP     PROC_A          ;intersegment jump
```

or

```
        MOV     AX,SEG SEG_A    ;segment address
        DD      LABEL_A         ;segment:offset pointer
```

Finally, .COM programs must not declare any segments with the STACK *combine* type. If a program declares a segment with the STACK *combine* type, the linker will insert initial SS and SP values into the .EXE file header, causing EXE2BIN to reject the .EXE file. A .COM program does not have explicitly declared stacks, although it can reserve space in a non-STACK *combine* type segment to which it can initialize the SP register *after* it receives control. The absence of a stack segment will cause the linker to issue a harmless warning message.

When the program is assembled and linked into a .EXE file, it must be converted into a binary file with a .COM extension by using the EXE2BIN utility as shown in the following example for the file YOURPROG.EXE:

```
C>EXE2BIN YOURPROG YOURPROG.COM  <Enter>
```

It is not necessary to delete or rename a .EXE file with the same filename as the .COM file before trying to execute the .COM file as long as both remain in the same directory, because MS-DOS's order of execution is .COM files first, then .EXE files, and finally .BAT files. However, the safest practice is to delete a .EXE file immediately after converting it to a .COM file in case the .COM file is later renamed or moved to a different directory. If a .EXE file designed for conversion to a .COM file is executed by accident, it is likely to crash the system.

Patching the .COM program using DEBUG

As discussed for .EXE files, a programmer who distributes software to users will probably want to send instructions on how to patch in error corrections. This approach to software updates lends itself even better to .COM files than it does to .EXE files.

For example, because .COM files contain only the code image, they need not be renamed in order to read and write them using DEBUG. The user need only be instructed on how to load the .COM file into DEBUG, how to patch the program, and how to write the patched image back to disk. Calculating the addresses and patch values is even easier, because no header exists in the .COM file image to cause complications. With the preceding exceptions, the details for patching .COM programs remain the same as previously outlined for .EXE programs.

.COM summary

To summarize, the .COM program and file structures are a simpler but more restricted approach to writing programs than the .EXE structure because the programmer has only a single memory model from which to choose (the .COM program segment model). Also, .COM program files do not contain the 512-byte (or more) header inherent to .EXE files, so the .COM program structure is well suited to small programs for which adding 512 bytes of header would probably at least double the file's size.

Summary of Differences

The following table summarizes the differences between .COM and .EXE programs.

	.COM program	.EXE program
Maximum size	65536 bytes minus 256 bytes for PSP and 2 bytes for stack	No limit
Entry point	PSP:0100H	Defined by END statement
CS at entry	PSP	Segment containing program's entry point
IP at entry	0100H	Offset of entry point within its segment
DS at entry	PSP	PSP
ES at entry	PSP	PSP
SS at entry	PSP	Segment with STACK attribute
SP at entry	FFFEH or top word in available memory, whichever is lower	End of segment defined with STACK attribute
Stack at entry	Zero word	Initialized or uninitialized, depending on source
Stack size	65536 bytes minus 256 bytes for PSP and size of executable code and data	Defined in segment with STACK attribute
Subroutine calls	NEAR	NEAR or FAR
Exit method	Interrupt 21H Function 4CH preferred; NEAR RET if MS-DOS versions 1.x	Interrupt 21H Function 4CH preferred; indirect jump to PSP:0000H if MS-DOS versions 1.x
Size of file	Exact size of program	Size of program plus header (at least 512 extra bytes)

Which format the programmer uses for an application usually depends on the program's intended size, but the decision can also be influenced by a program's need to address multiple memory segments. Normally, small utility programs (such as CHKDSK and FORMAT) are designed as .COM programs; large programs (such as the Microsoft C Compiler) are designed as .EXE programs. The ultimate decision is, of course, the programmer's.

Keith Burgoyne