CHAPTER 2

Organizing Segments

Understanding segments is an essential part of programming in assembly language. In the family of 8086-based processors, the term segment has two meanings:

- A block of memory of discrete size, called a “physical segment.” The number of bytes in a physical memory segment is 64K for 16-bit processors or 4 gigabytes for 32-bit processors.
- A variable-sized block of memory, called a “logical segment,” occupied by a program’s code or data.

As you read this chapter, the distinction between the two definitions will become clear. The adjectives “physical” and “logical” are not often used when speaking of segments. The beginning programmer is left to infer from context which definition applies. Fortunately, this is not difficult, and a distinction is often not required.

This chapter begins with a close look at physical memory segments. This lays the foundation for understanding logical segments, which form the subject of most of the following sections.

The section “Using Simplified Segment Directives” explains how to begin, end, and organize segments. It also explains how to access far data and code with simplified segment directives.

The next section, “Using Full Segment Definitions,” describes how to order, combine, and divide segments, and how to use the **SEGMENT** directive to define full segments. It also explains how to create a segment group so that you can use one segment address to access all the data.

Most of the information in this chapter also applies to writing modules to be called from other programs. Exceptions are noted when they apply. For more information about multiple-module programming, see Chapter 8, “Sharing Data and Procedures Among Modules and Libraries.”

Physical Memory Segments

As explained in Chapter 1, a physical segment can begin only at memory locations evenly divisible by 16, including address 0. Intel calls such locations "paragraphs." You can easily recognize a paragraph location because its hexadecimal address always ends with 0, as in 10000h or 2EA70h. The 8086/286 processors allow segments 64K in size, the largest number 16 bits can represent. The 80386/486 processors still adhere to the 64K limit when running in real mode. In protected mode, however, they use 32-bit registers that can hold addresses up to 4 gigabytes.

Segmented architecture presents certain hurdles for the assembly-language programmer. For small programs, the limitations lose importance. Code and data each occupy less than 64K and reside in individual segments. A simple offset locates each variable or instruction within a segment.

Larger programs, however, must contend with problems of segmented memory areas. If data occupies two or more segments, the program must specify both segment and offset to access a variable. When the data forms a continuous stream across segments—such as the text in a word processor's workspace—the problems become more acute. Whenever it adds or deletes text in the first segment, the word processor must seamlessly move data back and forth over the boundaries of each following segment.

The problem of segment boundaries disappears in the so-called flat address space of 32-bit protected mode. Although segments still exist, they easily hold all the code and data of the largest programs. Even a very large program becomes in effect a small application, able to reach all code and data with a single offset address.

Logical Segments

Logical segments contain the three components of a program: code, data, and stack. MASM organizes the three parts for you so they occupy physical segments of memory. The segment registers CS, DS, and SS contain the addresses of the physical memory segments where the logical segments reside.

You can define segments in two ways: with simplified segment directives and with full segment definitions. You can also use both kinds of segment definitions in the same program.

Simplified segment directives hide many of the details of segment definition and assume the same conventions used by Microsoft high-level languages. (See the following section, "Using Simplified Segment Directives.") The simplified segment directives generate necessary code, specify segment attributes, and arrange segment order.

Full segment definitions require more complex syntax but provide more complete control over how the assembler generates segments. (See "Using Full Segment Definitions" later in this chapter.) If you use full segment definitions, you must write code to handle all the tasks performed automatically by the simplified segment directives.

Using Simplified Segment Directives

Structuring a MASM program using simplified segments requires use of several directives to assign standard names, alignment, and attributes to the segments in your program. These directives define the segments in such a way that linking with Microsoft high-level languages is easy.

The simplified segment directives are **.MODEL**, **.CODE**, **.CONST**, **.DATA**, **.DATA?**, **.FARDATA**, **.FARDATA?**, **.STACK**, **.STARTUP**, and **.EXIT**. The following sections discuss these directives and the arguments they take.

MASM programs consist of modules made up of segments. Every program written only in MASM has one main module, where program execution begins. This main module can contain code, data, or stack segments defined with all of the simplified segment directives. Any additional modules should contain only code and data segments. Every module that uses simplified segments must, however, begin with the **.MODEL** directive.

The following example shows the structure of a main module using simplified segment directives. It uses the default processor (8086) and the default stack distance (**NEARSTACK**). Additional modules linked to this main program would use only the **.MODEL**, **.CODE**, and **.DATA** directives and the **END** statement.

```
; This is the structure of a main module
; using simplified segment directives

        .MODEL small, c ; This statement is required before you
                        ;   can use other simplified segment directives

        .STACK          ; Use default 1-kilobyte stack

        .DATA           ; Begin data segment

                        ; Place data declarations here

        .CODE           ; Begin code segment
        .STARTUP        ; Generate start-up code

                        ; Place instructions here

        .EXIT           ; Generate exit code
        END
```

The **.DATA** and **.CODE** statements do not require any separate statements to define the end of a segment. They close the preceding segment and then open a new segment. The **.STACK** directive opens and closes the stack segment but does not close the current segment. The **END** statement closes the last segment and marks the end of the source code. It must be at the end of every module.

Defining Basic Attributes with .MODEL

The **.MODEL** directive defines the attributes that affect the entire module: memory model, default calling and naming conventions, operating system, and stack type. This directive enables use of simplified segments and controls the name of the code segment and the default distance for procedures.

You must place **.MODEL** in your source file before any other simplified segment directive. The syntax is:

.MODEL *memorymodel* [[, *modeloptions*]]

The *memorymodel* field is required and must appear immediately after the **.MODEL** directive. The use of *modeloptions*, which define the other attributes, is optional. The *modeloptions* must be separated by commas. You can also use equates passed from the ML command line to define the *modeloptions*.

The following list summarizes the *memorymodel* field and the *modeloptions* fields, which specify language and stack distance:

Field	Description
Memory model	**TINY**, **SMALL**, **COMPACT**, **MEDIUM**, **LARGE**, **HUGE**, or **FLAT**. Determines size of code and data pointers. This field is required.
Language	**C**, **BASIC**, **FORTRAN**, **PASCAL**, **SYSCALL**, or **STDCALL**. Sets calling and naming conventions for procedures and public symbols.
Stack distance	**NEARSTACK** or **FARSTACK**. Specifying **NEARSTACK** groups the stack segment into a single physical segment (DGROUP) along with data. SS is assumed to equal DS. **FARSTACK** does not group the stack with DGROUP; thus SS does not equal DS.

You can use no more than one reserved word from each field. The following examples show how you can combine various fields:

```
.MODEL   small                 ; Small memory model
.MODEL   large, c, farstack    ; Large memory model,
                               ;   C conventions,
                               ;   separate stack
.MODEL   medium, pascal        ; Medium memory model,
                               ;   Pascal conventions,
                               ;   near stack (default)
```

The next four sections give more detail on each field.

Defining the Memory Model

MASM supports the standard memory models used by Microsoft high-level languages—tiny, small, medium, compact, large, huge, and flat. You specify the memory model with attributes of the same name placed after the **.MODEL** directive. With the exception of the flat model, which requires instructions specific to the 80386/486, your choice of a memory model does not limit the kind of instructions you can write. The memory model does, however, control segment defaults and determine whether data and code are near or far by default, as indicated in the following table.

Table 2.1 Attributes of Memory Models

Memory Model	Default Code	Default Data	Operating System	Data and Code Combined
Tiny	Near	Near	MS-DOS	Yes
Small	Near	Near	MS-DOS, Windows	No
Medium	Far	Near	MS-DOS, Windows	No
Compact	Near	Far	MS-DOS, Windows	No
Large	Far	Far	MS-DOS, Windows	No
Huge	Far	Far	MS-DOS, Windows	No
Flat	Near	Near	Windows NT	Yes

When writing assembler modules for a high-level language, you should use the same memory model as the calling language. Choose the smallest memory model available that can contain your data and code, since near references operate more efficiently than far references.

The predefined symbol **@Model** returns the memory model, encoding memory models as integers 1 through 7. For more information on predefined symbols, see "Predefined Symbols" in Chapter 1. For an example of how to use them, see Help.

The seven memory models supported by MASM 6.1 fall into three groups, described in the following paragraphs.

Small, Medium, Compact, Large, and Huge Models

The traditional memory models recognized by many languages are small, medium, compact, large, and huge. Small model supports one data segment and one code segment. All data and code are near by default. Large model supports multiple code and multiple data segments. All data and code are far by default. Medium and compact models are in-between. Medium model supports multiple code and single data segments; compact model supports multiple data segments and a single code segment.

Huge model implies individual data items larger than a single segment, but the implementation of huge data items must be coded by the programmer. Since the assembler provides no direct support for this feature, huge model is essentially the same as large model.

In each of these models, you can override the default. For example, you can make large data items far in small model, or internal procedures near in large model.

Tiny Model

Tiny-model programs run only under MS-DOS. Tiny model places all data and code in a single segment. Therefore, the total program file size can occupy no more than 64K. The default is near for code and static data items; you cannot override this default. However, you can allocate far data dynamically at run time using MS-DOS memory allocation services.

Tiny model produces MS-DOS .COM files. Specifying `.MODEL tiny` automatically sends the /TINY argument to the linker. Therefore, the /AT argument is not necessary with `.MODEL tiny`. However, /AT does not insert a **.MODEL** directive. It only verifies that there are no base or pointer fixups, and sends /TINY to the linker.

Flat Model

The flat memory model is a nonsegmented configuration available in 32-bit operating systems. It is similar to tiny model in that all code and data go in a single 32-bit segment.

To write a flat model program, specify the **.386** or **.486** directive before **.MODEL FLAT**. All data and code (including system resources) are in a single 32-bit segment. The operating system automatically initializes segment registers at load time; you need to modify them only when mixing 16-bit and 32-bit segments in a single application. CS, DS, ES, and SS all occupy the supergroup **FLAT**. Addresses and pointers passed to system services are always 32-bit near addresses and pointers.

Choosing the Language Convention

The language option facilitates compatibility with high-level languages by determining the internal encoding for external and public symbol names, the code generated for procedure initialization and cleanup, and the order that arguments are passed to a procedure with **INVOKE**. It also facilitates compatibility with high-level–language modules. The **PASCAL**, **BASIC**, and **FORTRAN** conventions are identical. **C** and **SYSCALL** have the same calling convention but different naming conventions. Functions in the Windows API use the Pascal calling convention.

Procedure definitions **(PROC)** and high-level procedure calls (**INVOKE)** automatically generate code consistent with the calling convention of the specified language. The **PROC**, **INVOKE**, **PUBLIC**, and **EXTERN** directives all use the naming convention of the language. These directives follow the default language conventions from the **.MODEL** directive unless you specifically override the default. Use of these directives is explained in "Controlling Program Flow," Chapter 7. You can also use the **OPTION** directive to set the language type. (See "Using the OPTION Directive" in Chapter 1.) Not specifying a language type in either the **.MODEL**, **OPTION**, **EXTERN**, **PROC**, **INVOKE**, or **PROTO** statement causes the assembler to generate an error.

The predefined symbol **@Interface** provides information about the language parameters. For a description of the bit flags, see Help.

For more information on calling and naming conventions, see Chapter 12, "Mixed-Language Programming." For information about writing procedures and prototypes, see Chapter 7, "Controlling Program Flow." For information on multiple-module programming, refer to Chapter 8, "Sharing Data and Procedures Among Modules and Libraries."

Setting the Stack Distance

The **NEARSTACK** keyword places the stack segment in the group DGROUP along with the data segment. The **.STARTUP** directive then generates code to adjust SS:SP so that SS (Stack Segment register) holds the same address as DS (Data Segment register). If you do not use **.STARTUP**, you must make this adjustment or your program may fail to run. (For information about startup code, see "Starting and Ending Code with .STARTUP and .EXIT," later in this chapter.) In this case, you can use DS to access stack items (including parameters and local variables) and SS to access near data. Furthermore, since stack items share the same segment address as near data, you can reliably pass near pointers to stack items.

The **FARSTACK** setting gives the stack a segment of its own. That is, SS does not equal DS. The default stack type, **NEARSTACK**, is a convenient setting for

most programs. Use **FARSTACK** for special cases such as memory-resident programs

and dynamic-link libraries (discussed in Chapters 10 and 11) when you cannot assume that the caller's stack is near. You can use the predefined symbol **@Stack** to determine if the stack location is DGROUP (for near stacks) or STACK (for far stacks).

Specifying a Processor and Coprocessor

MASM supports a set of directives for selecting processors and coprocessors. Once you select a processor, you must use only the instruction set for that processor. The default is the 8086 processor. If you always want your code to run on this processor, you do not need to add any processor directives.

To enable a different processor mode and the additional instructions available on that processor, use the directives **.186**, **.286**, **.386**, and **.486**. The instruction timings on a listing (see Appendix C, "Generating and Reading Assembly Listings") correspond to whichever processor directive you select.

The **.286P**, **.386P**, and **.486P** directives enable the instructions available only at higher privilege levels in addition to the normal instruction set for the given processor. Generally, you don't need privileged instructions unless you are writing operating-systems code or device drivers.

In addition to enabling different instruction sets, the processor directives also affect the behavior of extended language features. For example, the **INVOKE** directive pushes arguments onto the stack. If the **.286** directive is in effect, **INVOKE** takes advantage of operations possible only on 80286 and later processors.

Use the directives **.8087** (the default), **.287**, **.387**, and **.NO87** to select a math coprocessor instruction set. The **.NO87** directive turns off assembly of all coprocessor instructions. Note that **.486** also enables assembly of all coprocessor instructions because the 80486 processor has a complete set of coprocessor registers and instructions built into the chip. The processor instructions imply the corresponding coprocessor directive. The coprocessor directives are provided to override the defaults.

Creating a Stack

The stack is the section of memory used for pushing or popping registers and storing the return address when a subroutine is called. The stack often holds temporary and local variables.

If your main module is written in a high-level language, that language handles the details of creating a stack. Use the **.STACK** directive only when you write a main module in assembly language.

The **.STACK** directive creates a stack segment. By default, the assembler allocates 1K of memory for the stack. This size is sufficient for most small programs.

To create a stack of a size other than the default size, give **.STACK** a single numeric argument indicating stack size in bytes:

```
.STACK  2048            ; Use 2K stack
```

For a description of how stack memory is used with procedure calls and local variables, see Chapter 7, "Controlling Program Flow."

Creating Data Segments

Programs can contain both near and far data. In general, you should place important and frequently used data in the near data area, where data access is faster. This area can get crowded, however, because in 16-bit operating systems the total amount of all near data in all modules cannot exceed 64K. Therefore, you may want to place infrequently used or particularly large data items in a far data segment.

The **.DATA**, **.DATA?**, **.CONST**, **.FARDATA**, and **.FARDATA?** directives create data segments. You can access the various segments within DGROUP without reloading segment registers (see "Defining Segment Groups," later in this chapter). These five directives also prevent instructions from appearing in data segments by assuming CS to **ERROR**.

Near Data Segments

The **.DATA** directive creates a near data segment. This segment contains the frequently used data for your program. It can occupy up to 64K in MS-DOS or 512 megabytes under flat model in Windows NT. It is placed in a special group identified as DGROUP, which is also limited to 64K.

When you use **.MODEL**, the assembler automatically defines DGROUP for your near data segment. The segments in DGROUP form near data, which can normally be accessed directly through DS or SS.

You can also define the **.DATA?** and **.CONST** segments that go into DGROUP unless you are using flat model. Although all of these segments (along with the stack) are eventually grouped together and handled as data segments, **.DATA?** and **.CONST** enhance compatibility with Microsoft high-level languages. In Microsoft languages, **.CONST** is used to define constant data such as strings and floating-point numbers that must be stored in memory. The **.DATA?** segment is used for storing uninitialized variables. You can follow this convention if you want. If you use C startup code, **.DATA?** is initialized to 0.

You can use **@data** to determine the group of the data segment and **@DataSize** to determine the size of the memory model set by the **.MODEL** directive. The predefined symbols **@WordSize** and **@CurSeg** return the size attribute and name of the current segment, respectively. See "Predefined Symbols" in Chapter 1.

Far Data Segments

The compact, large, and huge memory models use far data addresses by default. With these memory models, however, you can still construct data segments using **.DATA**, **.DATA?**, and **.CONST**. The effect of these directives does not change from one memory model to the next. They always contribute segments to the default data area, DGROUP, which has a total limit of 64K.

When you use **.FARDATA** or **.FARDATA?** in the small and medium memory models, the assembler creates far data segments FAR_DATA and FAR_BSS, respectively. You can access variables with:

```
mov     ax, SEG farvar2
mov     ds, ax
```

For more information on far data, see "Near and Far Addresses" in Chapter 3.

Creating Code Segments

Whether you are writing a main module or a module to be called from another module, you can have both near and far code segments. This section explains how to use near and far code segments and how to use the directives and predefined equates that relate to code segments.

Near Code Segments

The small memory model is often the best choice for assembly programs that are not linked to modules in other languages, especially if you do not need more than 64K of code. This memory model defaults to near (two-byte) addresses for code and data, which makes the program run faster and use less memory.

When you use **.MODEL** and simplified segment directives, the **.CODE** directive in your program instructs the assembler to start a code segment. The next segment directive closes the previous segment; the **END** directive at the end of your program closes remaining segments. The example at the beginning of "Using Simplified Segment Directives," earlier in this chapter, shows how to do this.

You can use the predefined symbol **@CodeSize** to determine whether code pointers default to **NEAR** or **FAR**.

Far Code Segments

When you need more than 64K of code, use the medium, large, or huge memory model to create far segments.

The medium, large, and huge memory models use far code addresses by default. In the larger memory models, the assembler creates a different code segment for each module. If you use multiple code segments in the small,

compact, or tiny model, the linker combines the **.CODE** segments for all modules into one segment.

For far code segments, the assembler names each code segment MODNAME_TEXT, in which MODNAME is the name of the module. With near code, the assembler names every code segment _TEXT, causing the linker to concatenate these segments into one. You can override the default name by providing an argument after **.CODE**. (For a complete list of segment names generated by MASM, see Appendix E, "Default Segment Names.")

With far code, a single module can contain multiple code segments. The **.CODE** directive takes an optional text argument that names the segment. For instance, the following example creates two distinct code segments, FIRST_TEXT and SECOND_TEXT.

```
.CODE   FIRST
.
.       ; First set of instructions here
.
.CODE   SECOND
.
.       ; Second set of instructions here
.
```

Whenever the processor executes a far call or jump, it loads CS with the new segment address. No special action is necessary other than making sure that you use far calls and jumps. See "Near and Far Addresses" in Chapter 3.

Note The assembler always assumes that the CS register contains the address of the current code segment or group.

Starting and Ending Code with .STARTUP and .EXIT

The easiest way to begin and end an MS-DOS program is to use the **.STARTUP** and **.EXIT** directives in the main module. The main module contains the starting point and usually the termination point. You do not need these directives in a module called by another module.

These directives make MS-DOS programs easy to maintain. They automatically generate code appropriate to the stack distance specified with **.MODEL**. However, they do not apply to flat-model programs written for 32-bit operating systems. Thus, you should not use **.STARTUP** or **.EXIT** in programs written for Windows NT.

To start a program, place the **.STARTUP** directive where you want execution to begin. Usually, this location immediately follows the **.CODE** directive:

```
.CODE
.STARTUP
.
.               ; Place executable code here
.
.EXIT
END
```

Note that **.EXIT** generates executable code, while **END** does not. The **END** directive informs the assembler that it has reached the end of the module. All modules must end with the **END** directive whether you use simplified or full segments.

If you do not use **.STARTUP**, you must give the starting address as an argument to the **END** directive. For example, the following fragment shows how to identify a program's starting instruction with the label `start`:

```
        .CODE
start:
        .
        .               ; Place executable code here
        .
        END     start
```

Only the **END** directive for the module with the starting instruction should have an argument. When **.STARTUP** is present, the assembler ignores any argument to **END**.

For the default **NEARSTACK** attribute, **.STARTUP** points DS to DGROUP and sets SS:SP relative to DGROUP, generating the following code:

```
@Startup:
        mov     dx, DGROUP
        mov     ds, dx
        mov     bx, ss
        sub     bx, dx
        shl     bx, 1           ; If .286 or higher, this is
        shl     bx, 1           ;   shortened to shl bx, 4
        shl     bx, 1
        shl     bx, 1
        cli                     ; Not necessary in .286 or higher
        mov     ss, dx
        add     sp, bx
        sti                     ; Not necessary in .286 or higher
        .
        .
        .
        END     @Startup
```

An MS-DOS program with the **FARSTACK** attribute does not need to adjust SS:SP, so **.STARTUP** just initializes DS, like this:

```
@Startup:
       mov     dx, DGROUP
       mov     ds, dx
       .
       .
       .
       END     @Startup
```

When the program terminates, you can return an exit code to the operating system. Applications that check exit codes usually assume that an exit code of 0 means no problem occurred, and that an exit code of 1 means an error terminated the program. The **.EXIT** directive accepts a 1-byte exit code as its optional argument:

```
        .EXIT   1               ; Return exit code 1
```

.EXIT generates the following code that returns control to MS-DOS, thus terminating the program. The return value, which can be a constant, memory reference, or 1-byte register, goes into AL:

```
        mov     al, value
        mov     ah, 04Ch
        int     21h
```

If your program does not specify a return value, **.EXIT** returns whatever value happens to be in AL.

Using Full Segment Definitions

If you need complete control over segments, you can fully define the segments in your program. This section explains segment definitions, including how to order segments and how to define the segment types.

If you write a program under MS-DOS without **.MODEL** and **.STARTUP**, you must initialize registers yourself and use the **END** directive to indicate the starting address. The Windows operating system does not require you to initialize registers, as described in Chapter 3. For a description of typical startup code, see "Controlling the Segment Order," later in this chapter.

Defining Segments with the SEGMENT Directive

A defined segment begins with the **SEGMENT** directive and ends with the **ENDS** directive:

name **SEGMENT** [[*align*]] [[**READONLY**]] [[*combine*]] [[*use*]] [[*'class'*]]
statements
name **ENDS**

The *name* defines the name of the segment. Within a module, all segment definitions with the same name are treated as though they reference the same segment. The linker also combines identically named segments from different modules unless the combine type is **PRIVATE**. In addition, segments can be nested.

The optional types that follow the **SEGMENT** directive give the linker and the assembler instructions on how to set up and combine segments. The optional types, which are explained in detail in the following sections, include:

Type	Description
align	Defines the memory boundary on which a new segment begins.
READONLY	Tells the assembler to report an error if it detects an instruction modifying any item in a **READONLY** segment.
combine	Determines how the linker combines segments from different modules when building executable files.
use (80386/486 only)	Determines the size of a segment. **USE16** indicates that offsets in the segment are 16 bits wide. **USE32** indicates 32-bit offsets.
class	Provides a class name for the segment. The linker automatically groups segments of the same class in memory.

Types can be specified in any order. You can specify only one attribute from each of these fields; for example, you cannot have two different *align* types.

You can close a segment and reopen it later with another **SEGMENT** directive. When you reopen a segment, you need only give the segment name. You cannot change the attributes of a segment once you have defined it.

Note The **PAGE** *align* type and the **PUBLIC** *combine* type are distinct from the **PAGE** and **PUBLIC** directives. The assembler distinguishes them by means of context.

Aligning Segments

The optional *align* type in the **SEGMENT** directive defines the range of memory addresses from which a starting address for the segment can be selected. The *align* type can be any of the following:

Align Type	Starting Address
BYTE	Next available byte address.
WORD	Next available word address.
DWORD	Next available doubleword address.
PARA	Next available paragraph address (16 bytes per paragraph). Default.
PAGE	Next available page address (256 bytes per page).

The linker uses the alignment information to determine the relative starting address for each segment. The operating system calculates the actual starting address when the program is loaded.

Making Segments Read-Only

The optional **READONLY** attribute is helpful when creating read-only code segments for protected mode, or when writing code to be placed in read-only memory (ROM). It protects against illegal self-modifying code.

The **READONLY** attribute causes the assembler to check for instructions that modify the segment and to generate an error if it finds any. The assembler generates an error if you attempt to write directly to a read-only segment.

Combining Segments

The optional *combine* type in the **SEGMENT** directive defines how the linker combines segments having the same name but appearing in different modules.

The *combine* type controls linker behavior, not assembler behavior. The *combine* types, which are described in full detail in Help, include:

Combine Type	Linker Action
PRIVATE	Does not combine the segment with segments from other modules, even if they have the same name. Default.
PUBLIC	Concatenates all segments having the same name to form a single, contiguous segment.
STACK	Concatenates all segments having the same name and causes the operating system to set SS:00 to the bottom and SS:SP to the top of the resulting segment. Data initialization is unreliable, as discussed following.
COMMON	Overlaps segments. The length of the resulting area is the length of the largest of the combined segments. Data initialization is unreliable, as discussed following.
MEMORY	Used as a synonym for the **PUBLIC** *combine* type.
AT *address*	Assumes *address* as the segment location. An **AT** segment cannot contain any code or initialized data, but is useful for defining structures or variables that correspond to specific far memory locations, such as a screen buffer or low memory. You cannot use the **AT** *combine* type in protected-mode programs.

Do not place initialized data in **STACK** or **COMMON** segments. With these *combine* types, the linker overlays initialized data for each module at the beginning of the segment. The last module containing initialized data writes over any data from other modules.

Note Normally, you should provide at least one stack segment (having **STACK** *combine* type) in a program. If no stack segment is declared, LINK displays a warning message. You can ignore this message if you have a specific reason for not declaring a stack segment. For example, you would not have a separate stack segment in a MS-DOS tiny model (.COM) program, nor would you need a separate stack in a DLL that uses the caller's stack.

Setting Segment Word Sizes (80386/486 Only)

The *use* type in the **SEGMENT** directive specifies the segment word size on the 80386/486 processors. Segment word size determines the default operand and address size of all items in a segment.

The size attribute can be **USE16**, **USE32**, or **FLAT**. If you specify the **.386** or **.486** directive before the **.MODEL** directive, **USE32** is the default. This attribute specifies that items in the segment are addressed with a 32-bit offset rather than a

16-bit offset. If **.MODEL** precedes the **.386** or **.486** directive, **USE16** is the default. To make **USE32** the default, put **.386** or **.486** before **.MODEL**. You can override the **USE32** default with the **USE16** attribute, or vice versa.

Note Programs written for MS-DOS must not specify **USE32**. Mixing 16-bit and 32-bit segments in the same program is possible but usually applies only to systems programming.

Setting Segment Order with Class Type

The optional *class* type in the **SEGMENT** directive helps control segment ordering. Two segments with the same name are not combined if their class is different. The linker arranges segments so that all segments identified with a given *class* type are next to each other in the executable file. However, within a particular class, the linker arranges segments in the order encountered. The **.ALPHA**, **.SEQ**, or **.DOSSEG** directive determines this order in each .OBJ file. The most common method for specifying a *class* type is to place all code segments first in the executable file.

Controlling the Segment Order

The assembler normally positions segments in the object file in the order in which they appear in source code. The linker, in turn, processes object files in the order in which they appear on the command line. Within each object file, the linker outputs segments in the order they appear, subject to any group, class, and .**DOSSEG** requirements.

You can usually ignore segment ordering. However, it is important whenever you want certain segments to appear at the beginning or end of a program or when you make assumptions about which segments are next to each other in memory. For tiny model (.COM) programs, code segments must appear first in the executable file, because execution must start at the address 100h.

Segment Order Directives

You can control the order in which segments appear in the executable program with three directives. The default, **.SEQ**, arranges segments in the order in which you declare them.

The **.ALPHA** directive specifies alphabetical segment ordering within a module. **.ALPHA** is provided for compatibility with early versions of the IBM assembler. If you have trouble running code from older books on assembly language, try using **.ALPHA**.

The **.DOSSEG** directive specifies the MS-DOS segment-ordering convention. It places segments in the standard order required by Microsoft languages. Do not use **.DOSSEG** in a module to be called from another module.

The **.DOSSEG** directive orders segments as follows:

1. Code segments
2. Data segments, in this order:
   a. Segments not in class BSS or STACK
   b. Class BSS segments
   c. Class STACK segments

When you declare two or more segments to be in the same class, the linker automatically makes them contiguous. This rule overrides the segment-ordering directives. (For more about segment classes, see "Setting Segment Order with Class Type" in the previous section.)

Linker Control

Most of the segment-ordering techniques (class names, **.ALPHA**, and **.SEQ**) control the order in which the assembler outputs segments. Usually, you are more interested in the order in which segments appear in the executable file. The linker controls this order.

The linker processes object files in the order in which they appear on the command line. Within each module, it then outputs segments in the order given in the object file. If the first module defines segments DSEG and STACK and the second module defines CSEG, then CSEG is output last. If you want to place CSEG first, there are two ways to do so.

The simpler method is to use **.DOSSEG**. This directive is output as a special record to the object file linker, and it tells the linker to use the Microsoft segment-ordering convention. This convention overrides command-line order of object files, and it places all segments of class `'CODE'` first. (See "Defining Segments with the SEGMENT Directive," previous.)

The other method is to define all the segments as early as possible (in an include file, for example, or in the first module). These definitions can be "dummy segments"—that is, segments with no content. The linker observes the segment ordering given, then later combines the empty segments with segments in other modules that have the same name.

For example, you might include the following at the start of the first module of your program or in an include file:

```
_TEXT   SEGMENT WORD PUBLIC 'CODE'
_TEXT   ENDS
_DATA   SEGMENT WORD PUBLIC 'DATA'
_DATA   ENDS
CONST   SEGMENT WORD PUBLIC 'CONST'
CONST   ENDS
STACK   SEGMENT PARA STACK 'STACK'
STACK   ENDS
```

Later in the program, the order in which you write `_TEXT`, `_DATA`, or other segments does not matter because the ultimate order is controlled by the segment order defined in the include file.

Setting the ASSUME Directive for Segment Registers

Many of the assembler instructions assume a default segment. For example, **JMP** assumes the segment associated with the CS register, **PUSH** and **POP** assume the segment associated with the SS register, and **MOV** instructions assume the segment associated with the DS register.

When the assembler needs to reference an address, it must know what segment contains the address. It finds this by using the default segment or group addresses assigned with the **ASSUME** directive. The syntax is:

ASSUME segregister : seglocation [, segregister : seglocation]]
ASSUME dataregister : qualifiedtype [, dataregister : qualifiedtype]
ASSUME register : **ERROR** [, register : **ERROR**]
ASSUME [register :] **NOTHING** [, register : **NOTHING**]
ASSUME register : **FLAT** [, register : **FLAT**]

The *seglocation* must be the name of the segment or group that is to be associated with *segregister*. Subsequent instructions that assume a default register for referencing labels or variables automatically assume that if the default segment is *segregister*, the label or variable is in the *seglocation*. MASM 6.1 automatically gives CS the address of the current code segment. Therefore, you do not need to include

```
ASSUME CS : MY_CODE
```

at the beginning of your program if you want the current segment associated with CS.

Note Using the **ASSUME** directive to tell the assembler which segment to associate with a segment register is not the same as telling the processor. The **ASSUME** directive affects only assembly-time assumptions. You may need to use instructions to change run-time conditions. Initializing segment registers at run time is discussed in "Informing the Assembler About Segment Values," Chapter 3.

The **ASSUME** directive can define a segment for each of the segment registers. The *segregister* can be CS, DS, ES, or SS (and FS and GS on the 80386/486). The *seglocation* must be one of the following:

- The name of a segment defined in the source file with the **SEGMENT** directive.
- The name of a group defined in the source file with the **GROUP** directive.
- The keyword **NOTHING**, **ERROR**, or **FLAT.**
- A **SEG** expression (see "Immediate Operands" in Chapter 3).
- A string equate (text macro) that evaluates to a segment or group name (but not a string equate that evaluates to a **SEG** expression).

It is legal to combine assumes to **FLAT** with assumes to specific segments. Combinations might be necessary in operating-system code that handles both 16- and 32-bit segments.

The keyword **NOTHING** cancels the current segment assumptions. For example, the statement **ASSUME NOTHING** cancels all register assumptions made by previous **ASSUME** statements.

Usually, a single **ASSUME** statement defines all four segment registers at the start of the source file. However, you can use the **ASSUME** directive at any point to change segment assumptions.

Using the **ASSUME** directive to change segment assumptions is often equivalent to changing assumptions with the segment-override operator (**:**). See "Direct Memory Operands" in Chapter 3. The segment-override operator is more convenient for one-time overrides. The **ASSUME** directive may be more convenient if previous assumptions must be overridden for a sequence of instructions.

However, in either case, your program must explicitly load a segment register with a segment address before accessing data within the segment. **ASSUME** only tells the assembler to assume that the register is correctly initialized; it does not by itself generate any code to load the register.

You can also prevent the use of a register with:

```
ASSUME  SegRegister : ERROR
```

The assembler generates an `ASSUME CS:ERROR` when you use simplified directives to create data segments, effectively preventing instructions or code labels from appearing in a data segment.

For more information about **ASSUME**, refer to "Defining Register Types with ASSUME" in Chapter 3.

Defining Segment Groups

A group is a collection of segments totalling not more than 64K in 16-bit mode. A program addresses a code or data item in the group relative to the beginning of the group.

A group lets you develop separate logical segments for different kinds of data and then combine these into one segment (a group) for all the data. Using a group can save you from having to continually reload segment registers to access different segments. As a result, the program uses fewer instructions and runs faster.

The most common example of a group is the specially named group for near data, DGROUP. In the Microsoft segment model, several segments (_DATA, _BSS, CONST, and STACK) are combined into a single group called DGROUP. Microsoft high-level languages place all near data segments in this group. (By default, the stack is placed here, too.) The **.MODEL** directive automatically defines DGROUP. The DS register normally points to the beginning of the group, giving you relatively fast access to all data in DGROUP.

The syntax of the group directive is:

name **GROUP** *segment* [[, *segment*]]...

The *name* labels the group. It can refer to a group that was previously defined. This feature lets you add segments to a group one at a time. For example, if `MYGROUP` was previously defined to include `ASEG` and `BSEG`, then the statement

```
MYGROUP GROUP   CSEG
```

is perfectly legal. It simply adds `CSEG` to the group `MYGROUP; ASEG` and `BSEG` are not removed.

Each *segment* can be any valid segment name (including a segment defined later in source code), with one restriction: a segment cannot belong to more than one group.

The **GROUP** directive does not affect the order in which segments of a group are loaded. You can place any number of 16-bit segments in a group as long as the total size does not exceed 65,536 bytes. If the processor is in 32-bit mode, the maximum size is 4 gigabytes. You need to make sure that non-grouped segments do not get placed between grouped segments in such a way that the size of the group exceeds 64K or 4 gigabytes. Neither can you place a 16-bit and a 32-bit segment in the same group.